Université de Nancy I
Centre de Recherche en
Informatique de Nancy

Supélec Metz
Service Intelligence
Artificielle et Productique

Rapport de D.E.A. en Informatique

PAPER
Une approche algorithmique cellulaire parallèle du perceptron multicouche

Présenté le 13 septembre 1993
par
François PARMENTIER

Membres du jury :

Didier GALMICHE
Monique GRANDBASTIEN
Claude LHERMITTE
Stéphane VIALLE
Frédéric ALEXANDRE

Remerciements

Nous tenons à remercier particulièrement :

- Stéphane VIALLE, pour nous avoir guidé et soutenu tout au long de notre travail ;
- Claude LHERMITTE, pour nous avoir accueilli au sein du service qu'il dirige à Supélec ;
- Yannick LALLEMENT, pour son aide et sa disponibilité ;
- tout le personnel du service IAP, pour son accueil chaleureux.

Table des matières

Table des figures

Liste des tableaux

Introduction

Le langage cellulaire exploitant la MCV [1] n'est pas encore très utilisé de nos jours. Nous avons voulu tester sa facilité d'utilisation en développant une application adaptée à sa philosophie et de taille respectable.

C'est pourquoi nous avons écrit une implantation d'un genre de réseau de neurones très utilisé (le perceptron multicouche). Elle devrait permettre d'automatiser facilement et naturellement l'analyse et la modification des réseaux de neurones.

Nous verrons dans la première partie les principes des réseaux de neurones, ceux de la MCV dans la seconde. La troisième partie nous montrera les problèmes d'algorithmique cellulaire soulevés par l'implantation du perceptron multicouche en MCV.

1. MCV pour Machine Cellulaire Virtuelle

Chapitre 1

Réseaux de neurones

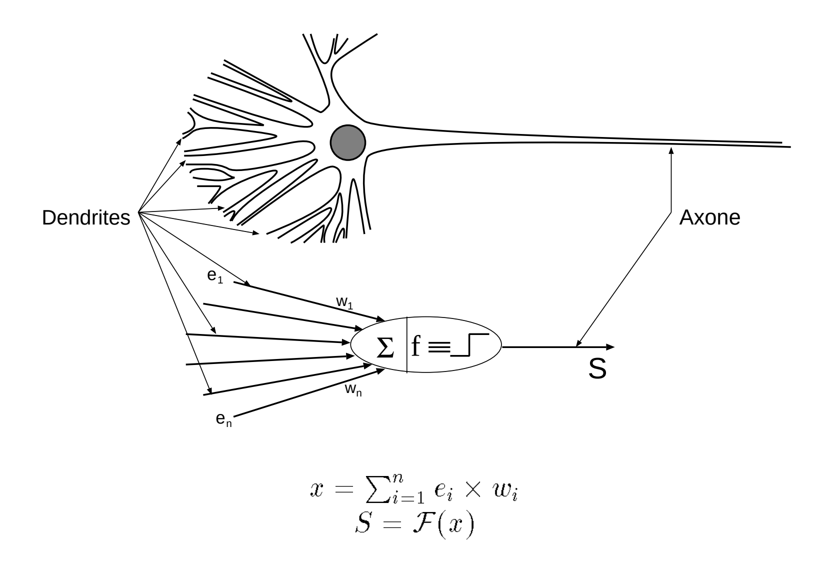
1.1 Introduction aux réseaux de neurones

Les réseaux de neurones ne datent pas d'hier, voici un petit rappel dû à Jean-Paul HATON [Hat89] :

> « L'idée des modèles connexionnistes ou *neuro-mimétiques* n'est pas nouvelle. En fait, dès le début de l'intelligence artificielle, deux courants de pensée se sont développés indépendamment et parfois concurremment, que l'on peut résumer par la phrase lapidaire de H.DREYFUS : *Making a mind versus modelling the brain.*
>
> « La première approche (*making a mind*) adoptée au départ par des chercheurs comme H.SIMON et A.NEWELL a conduit à l'I.A. actuelle et aux systèmes à bases de connaissances manipulant des données symboliques.
>
> « La seconde (*modelling the brain*) (...) Presque totalement abandonnée à la fin des années 60 après l'ouvrage célèbre de M.MINSKY et S.PAPERT de 1969, (...) est réapparue récemment avec une vigueur accrue. »

Par analogie avec le cerveau humain, les chercheurs en Intelligence Artificielle ont élaboré un modèle de neurone artificiel, inspiré des connaissances des neurobiologistes sur les neurones naturels. Un neurone artificiel est une unité de calcul munie de plusieurs entrées (dendrites) et d'une sortie (axone). Chaque neurone a un état interne, calculé comme la somme pondérée des valeurs d'entrée [1], seuillée par une fonction interne. Ce calcul effectué, le neurone propage son nouvel état interne sur son axone (*cf. figure 1.1*).

1. le poids d'une entrée dans la somme calculée dépend de la liaison considérée



$$x = \sum_{i=1}^{n} e_i \times w_i$$
$$S = \mathcal{F}(x)$$

FIG. 1.1 - *Analogie entre un neurone biologique et un neurone artificiel*

Les fonctions de seuillage se divisent toutes en trois parties (*cf. figure 1.2*) :

- une partie non-activée, en dessous du seuil ;
- une phase de transition, aux alentours du seuil ;
- une partie activée, au dessus du seuil.

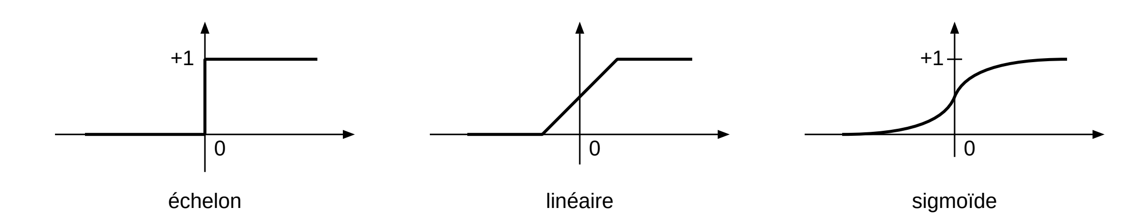


FIG. 1.2 - *Les trois principales fonctions de seuillage utilisées*

Un réseau de neurones est un ensemble de neurones connectés les uns aux autres par des liaisons axone-dendrite. On dit que le réseau de neurones passe d'un état à un autre lorsque tous ses neurones ont recalculé leur état interne, en fonction de leurs entrées. Ce processus itératif peut être effectué en séquentiel ou en parallèle.

L'ensemble des poids des liaisons détermine le fonctionnement du réseau de neurones. Les motifs sont présentés à un sous-ensemble du réseau de neurones : la couche d'entrée. Lorsqu'on applique un motif à un réseau, celui-ci cherche à atteindre un état stable. Lorqu'il est atteint, les valeurs d'activation des neurones de sortie constituent le résultat.

Les types de réseau de neurones diffèrent par plusieurs paramètres :

- la topologie des connections entre les neurones ;
- la fonction de seuillage utilisée (sigmoïde, échelon, fonction linéaire. . .) notée, par la suite, $\mathcal{F}$;

– l'algorithme d'apprentissage.

Un apprentissage est dit supervisé lorsque l'on force le réseau à converger vers un état final précis, en même temps qu'on lui présente un motif. À l'inverse, lors d'un apprentissage non-supervisé, le réseau est laissé libre de converger vers n'importe quel état final lorsqu'on lui présente un motif (*cf.* § *1.4*).

1.2 Le perceptron

1.2.1 Le perceptron monocouche

Le perceptron monocouche est un réseau de neurones à deux couches, une couche d'entrée et une couche de sortie. Ce perceptron est dit monocouche car il ne possède qu'une couche active (celle de sortie) par opposition aux perceptrons multicouches, qui ont plus d'une couche active. Chaque neurone de la couche d'entrée est relié à tous les neurones de la couche de sortie. (*cf. figure 1.3*).

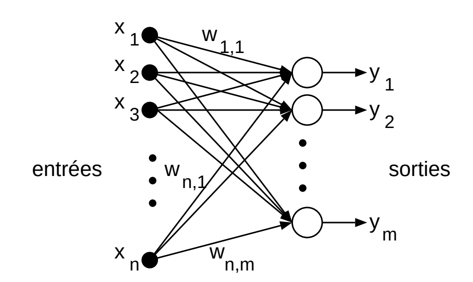


FIG. 1.3 - *Le perceptron monocouche*

L'apprentissage de ce réseau se fait comme suit (*cf. figure 1.5*). Tout d'abord les poids de toutes les liaisons sont initialisés avec de petites valeurs aléatoires, en général de l'ordre de $-0,5$ à $+0,5$. Puis un motif est présenté au réseau, lequel calcule la sortie correspondante. Les poids des liaisons sont ensuite mis à jour proportionnellement à l'écart entre la sortie calculée et la sortie désirée. Les motifs sont présentés les uns après les autres. L'ensemble des motifs est présenté plusieurs fois jusqu'à ce que les poids des liaisons soient stables [Lip87]. C'est un apprentissage supervisé.

La principale limitation du perceptron monocouche est que, dans l'espace des motifs, les limites des classes qu'il reconnaît sont des hyperplans. On dit qu'il ne peut résoudre que des problèmes linéairement séparables. (*cf. figure 1.4*).

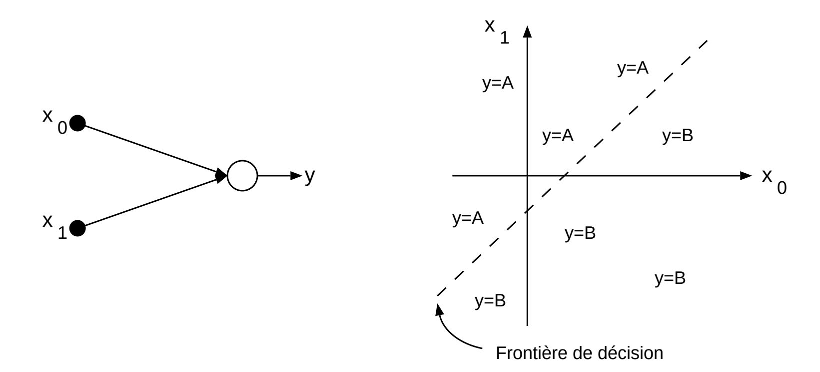


FIG. 1.4 - *Type de problème résolu par un perceptron monocouche à un neurone de sortie, discrimination des deux classes $y = A$ ou $y = B$*

1. initialisation des poids et du seuil

   Donner à $w_{ij}(0)$ $(1 \leq i \leq n, 1 \leq j \leq m)$ des valeurs aléatoires comprises, par exemple, entre $-0,5$ et $+0,5$, $w_{ij}(t)$ étant le poids de la liaison entre l'entrée i et le neurone de sortie j à l'instant t.

2. présentation d'une nouvelle entrée et de la sortie désirée

   Présenter un vecteur d'entrée $x_1(t), \ldots, x_n(t)$ et le vecteur de sortie désiré $d_1(t), \ldots, d_m(t)$

3. calcul de la sortie

$$y_j(t) \; = \; \mathcal{F}\left(\sum_{i=1}^{n} w_{ij}(t)x_i(t)\right)$$

4. mise à jour des poids

$$w_{ij}(t+1) \; = \; w_{ij}(t) \; + \; \eta\left[d_j(t) \; - \; y_j(t)\right] x_i(t) \quad (1 \leq i \leq n, 1 \leq j \leq m)$$

   où η est un facteur de gain positif et inférieur à 1

5. retourner en 2 jusqu'à la convergence (c'est-à-dire $d_j(t) \simeq y_j(t)$)

FIG. 1.5 - *Algorithme d'apprentissage du perceptron monocouche (avec n neurones d'entrée et m neurones de sortie)*

1.2.2 Le perceptron multicouche

Ce type de perceptron comporte une ou plusieurs couches cachées entre la couche d'entrée et la couche de sortie. Ces couches supplémentaires sont connectées aux couches

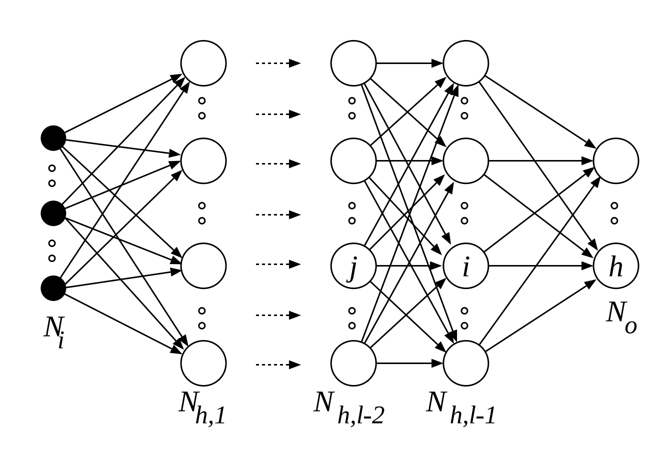


FIG. 1.6 - *Perceptron multicouche à $l-1$ couches cachées.*

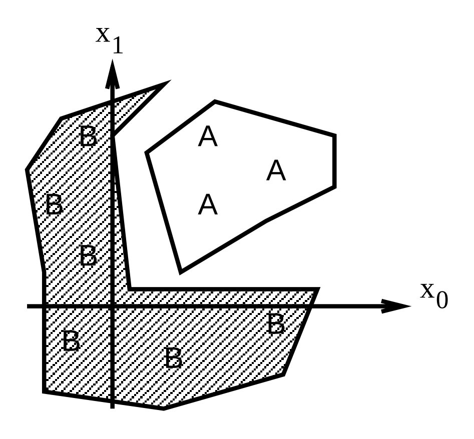


FIG. 1.7 - *Type de discrimination obtenue avec un réseau à 2 couches cachées avec 2 neurones de sortie : A et B, et 2 neurones d'entrée : x_0 et x_1*

voisines (*cf. figure 1.6*). Ce type de réseau résoud les problèmes qui mettent en échec le perceptron monocouche. Les perceptrons multicouches ont longtemps été délaissés car on ne connaissait pas d'algorithme d'apprentissage efficace. À l'heure actuelle, l'algorithme de rétro-propagation du gradient en est un.

L'algorithme d'apprentissage, dit de rétro-propagation du gradient, est analogue à celui du perceptron monocouche (*cf. figure 1.8*). La différence se situe au niveau de la mise à jour des poids des liaisons. Elle dépend, pour un neurone de sortie, de l'écart entre la valeur calculée et la sortie désirée, pour un neurone d'une couche cachée, de la somme pondérée des mises à jour des poids de la couche suivante, et, dans les deux cas, de l'état du neurone considéré [Coc88]. Cette mise à jour se fait donc récursivement en partant de la couche de sortie vers la couche d'entrée. Il faut noter que, dans ce réseau, la fonction de seuillage doit être dérivable ; on utilise habituellement une sigmoïde (*cf. figure 1.2*).

L'avantage de ce réseau par rapport au perceptron monocouche est que la limite des classes qu'il sait reconnaître devient convexe dans l'espace des motifs, s'il ne comporte qu'une couche cachée et quelconque s'il en comporte au moins deux (*cf. figure 1.7*). En utilisant plus de deux couches cachées, on ne change pas le type de problème résolvable, mais uniquement la vitesse de convergence de l'algorithme [Lip87]. Bien qu'aucune preuve de convergence de l'algorithme de rétro-propagation n'existe, ce type de réseau est l'un des plus employé actuellement car ses performances sont généralement bonnes. Néanmoins, le problème du dimensionnement des couches intermédiaires reste ouvert et il n'est résolu que de façon empirique (généralement par essai-modification) : si on augmente les tailles des couches cachées, le réseau apprend par cœur, si on les diminue, on le force à généraliser ([Coc88]) .

Cet algorithme cherche à minimiser l'erreur quadratique moyenne entre la sortie calculée et la sortie désirée. On utilise habituellement comme fonction de seuillage une sigmoïde :

$$\mathcal{F}(\alpha) = \frac{1}{1 + e^{-\alpha}}$$

dont les valeurs de sortie sont entre 0 et 1 (*cf. figure 1.2*).

1. initialisation des poids

   Donner à tous les poids une valeur aléatoire (p.ex. entre $-0,5$ et $+0,5$).

2. présentation d'une nouvelle entrée et de la sortie désirée

   Présenter le vecteur d'entrée $x_1, \ldots, x_n$ et le vecteur de sortie désirée correspondante $d_1, \ldots, d_m$.

3. calcul de la sortie de chaque neurone j

   $$y_j(t) = \mathcal{F}\left(\sum_{i=1}^{k} w_{ij}(t) y_i(t)\right)$$

   on calcule ceci successivement pour chaque couche du réseau, de l'entrée vers la sortie. k étant le nombre de neurones de la couche précédente.

4. mise à jour des poids, récursivement de la sortie vers l'entrée

   Chaque poids est modifié d'une valeur Δw_{ij} :

   $$\Delta w_{ij} = \eta \delta_j y_i(t)$$

   $$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$$

   où $w_{ij}(t)$ est le poids d'une liaison entre deux neurones i et j de deux couches consécutives au temps t, où η est un facteur de gain positif et inférieur à 1, où $y_i(t)$ est l'état du neurone i ou d'une entrée et δ_j :

   si j est un neurone de sortie alors

   $$\delta_j = 2 y_j (1 - y_j)(d_j - y_j)$$

   si j est un neurone caché alors

   $$\delta_j = y_j (1 - y_j) \sum_k \delta_k w_{jk}$$

   où k parcourt les neurones de la couche suivant celle du neurone j

5. retourner en 2 jusqu'à la convergence (idem monocouche)

FIG. 1.8 - *Algorithme d'apprentissage par rétro-propagation*

1.3 Le réseau de Hopfield

Ce type de réseau a été proposé par HOPFIELD en 1982. Il ne comporte qu'une seule couche ; chaque neurone est relié à tous les autres : pour N neurones il y a donc N^2 liaisons. Ses entrées sont de type binaire : entre -1 et $+1$. Chaque neurone joue à la fois le rôle d'entrée et de sortie (*cf. figure 1.9*).

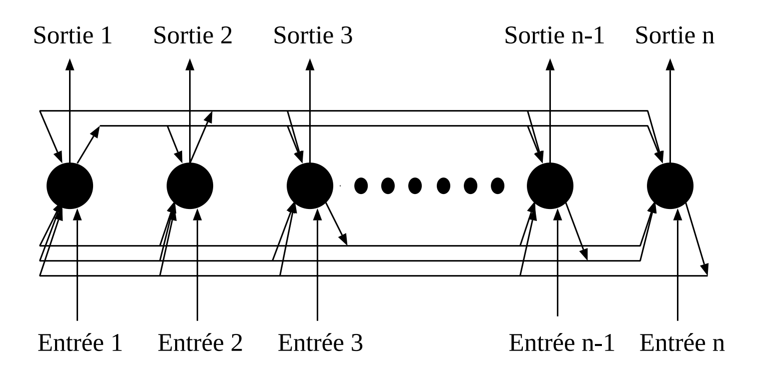


FIG. 1.9 - *Le réseau de Hopfield*

Lors de l'apprentissage (*cf. figure 1.11*) on fixe les poids des liaisons du réseau en fonction des motifs à reconnaître. On peut utiliser pour cela la règle de Hebb qui consiste à renforcer les connexions entre les neurones pour lesquels on souhaite des sorties de même signe, et à diminuer les poids des connexions dans le cas contraire. Les poids sont calculés de telle sorte que les motifs soient des états stables du réseau. Lorsqu'ensuite on lui présente un motif bruité ou incomplet, le réseau de Hopfield converge vers l'état stable le plus proche, qu'on espère être un motif appris. Le réseau de Hopfield fonctionne donc comme une mémoire associative. La fonction de seuillage utilisée est un échelon [2] dont les sorties sont comprises entre -1 et $+1$.

Il a deux principales limitations. La première est qu'il ne peut apprendre qu'un nombre limité de motifs. Si on essaie de lui en faire apprendre un trop grand nombre, le réseau de Hopfield peut alors converger vers un état parasite, qui n'est pas un des motifs à reconnaître. Le nombre de motifs à reconnaître acceptable est de l'ordre de 0,15 fois le nombre total de neurones du réseau [Lip87].

La figure 1.10 représente de manière imagée un réseau de Hopfield. Les trous représentent les états stables. Un motif quelconque correspond à un point du plan. La convergence est analogue au chemin d'une bille vers un trou. À gauche, les trous sont peu nombreux et bien espacés : il n'y a pas de problème. À droite, les motifs appris sont trop nombreux ; il apparaît des trous parasites issus de l'interférence entre deux motifs proches, et certains « bons » trous sont noyés dans leurs voisins.

La seconde limitation est qu'un motif peut ne pas devenir un état stable s'il est trop proche d'un autre motif, au sens de la distance de Hamming [3].

2. ou fonction de Heavyside
3. la distance de Hamming séparant deux mots binaires est le nombre de bits qui diffèrent

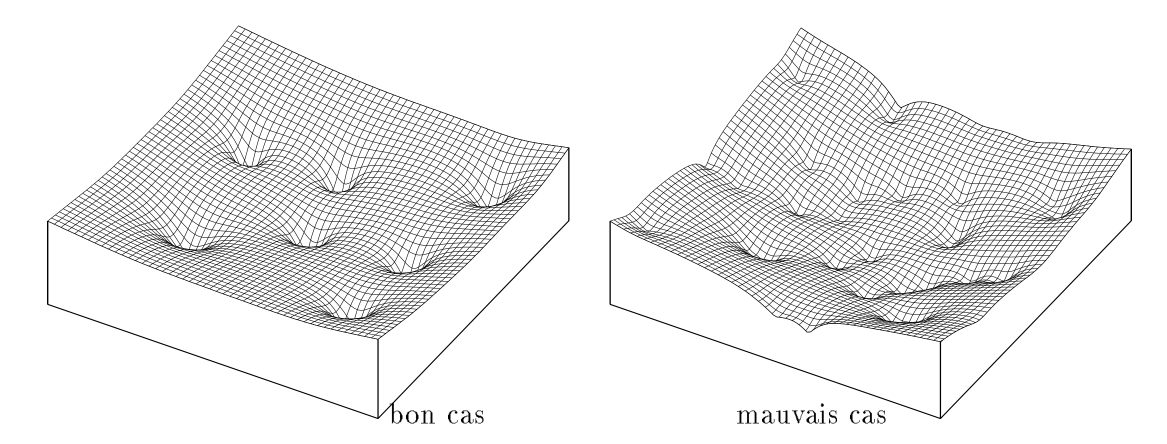


FIG. 1.10 - *Représentation imagée d'un réseau de Hopfield*

1. initialisation des poids des liaisons

$$t_{ij} = \begin{cases} \sum_{s=1}^{n} x_i^s x_j^s, & i \neq j \\ 0, & i = j \end{cases}$$

où t_{ij} est le poids de la liaison entre le neurone i et le neurone j et où x_i^s est l'élément i du motif de classe s.

2. présentation d'un motif inconnu

$$\mu_i(0) = x_i, \quad 1 \leq i \leq n$$

où $\mu_i(t)$ est la sortie du neurone i au temps t et où x_i est l'élément i du motif présenté.

3. mise à jour des poids : $t_{ij} = t_{ij} + \eta x_i x_j$ (règle de Hebb)

$$\mu_j(t+1) = \mathcal{F}\left(\sum_{i=1}^{n} t_{ij}\mu_i(t)\right), \quad 1 \leq j \leq n$$

$\mathcal{F}$ est une fonction en échelon.

4. répéter en retournant en 3 jusqu'à convergence

L'état final des neurones représente alors le motif appris le plus proche du motif présenté.

FIG. 1.11 - *Algorithme d'apprentissage du réseau de Hopfield*

1.4 Le réseau de Kohonen

Le réseau de Kohonen comporte deux couches : une couche d'entrée et une couche de sortie. Chaque neurone d'entrée est connecté à tous les neurones de la couche de sortie. Les neurones de la couche de sortie sont organisés en une grille à deux dimensions (*cf. figure 1.12*).

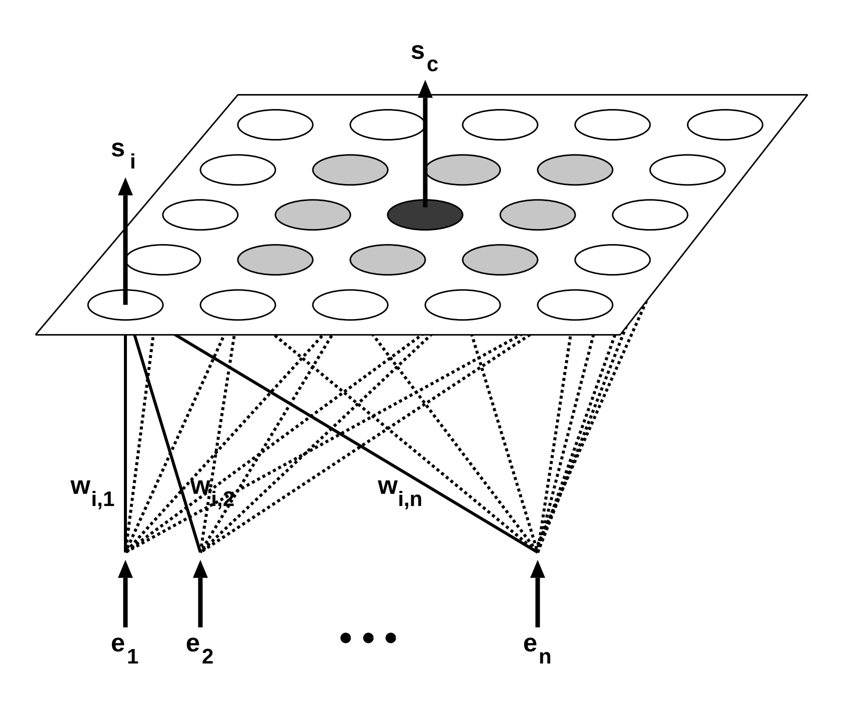


FIG. 1.12 - *Le réseau de Kohonen*

L'idée principale du réseau de Kohonen est de faire en sorte que les neurones se regroupent en aires, chacune étant chargée de reconnaître un motif. Les aires ne sont pas déterminées à l'avance mais apparaissent lors de l'apprentissage (*cf. figure 1.13*). Cet apprentissage est non-supervisé.

Les poids des liaisons sont initialisés à des valeurs aléatoires de l'ordre de $-0,5$ à $0,5$, puis chaque motif à classer est présenté. Après avoir présenté un motif au réseau, on détermine quel neurone de sortie est le plus proche de l'entrée, au sens d'une fonction de distance qui tient compte des poids des liaisons et de l'état des neurones de sortie. Ensuite les poids des liaisons partant de ce neurone sont mis à jour, ainsi que ceux des liaisons des neurones voisins. Puis on passe au motif suivant. L'ensemble des motif est présenté plusieurs fois jusqu'à ce que les poids des liaisons soient stables.

Les réseaux de Kohonen permettent de classifier des éléments lorsque l'on n'en connaît pas de partition *a priori*. C'est le principal avantage de l'apprentissage non-supervisé.

1. initialisation des poids

   Donner à tous les poids des liaisons de petites valeurs aléatoires.

2. présentation d'une nouvelle entrée

3. calcul de la distance à tous les nœuds

$$d_j = \sum_{i=1}^{n}(x_i(t) - w_{ij}(t))^2$$

   où $x_i(t)$ est l'état du neurone d'entrée i au temps t et $w_{ij}(t)$ le poids de la liaison entre l'entrée i et la sortie j au temps t.

4. sélection du neurone ayant la distance minimale

   Chercher le neurone $j^\star$ ayant le d_j minimal.

5. mise à jour des poids des liaisons du neurone $j^\star$ et de ses voisins avec la couche d'entrée

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t))$$

   où $\eta(t)$ est un terme de gain inférieur à 1 et positif qui décroît au cours du temps.

6. retourner en 2 jusqu'à la convergence

FIG. 1.13 - *Algorithme d'apprentissage du réseau de Kohonen*

1.5 Bibliographie

An Introduction to Computing with Neural Nets [Lip87] est un article de présentation. Il compare les classifieurs classiques aux réseaux de neurones. Il explique le réseau de Hopfield, les perceptrons mono et multicouche, les réseaux de Kohonen, et d'autres comme le réseau de Hamming, le classifieur de Carpenter-Grossberg ;

Réseaux de neurones [Coc88] est un article de synthèse présentant entre autres les implantations des algorithmes des réseaux de neurones. Il contient l'algorithme précis des réseaux multicouches, les principes du modèle de Hopfield, de la machine de Boltzmann, et une présentation de l'application NetTalk ;

Intelligence Artificielle et Connexionnisme [Lhe91] Article général, met en évidence les limites des approches symbolique et connexionniste. Conclut sur la complémentarité des deux approches et leur convergence ;

Modèles connexionnistes pour l'intelligence artificielle [Hat89] article de synthèse. Présentation et historique de l'approche neuronale. Discussion sur l'opposition symbolisme/connexionnisme. Introduction de la colonne corticale développée par l'équipe Cortex (de Nancy) et de la nécessité d'une architecture parallèle ;

Les machines neuronales [PDG88] article de vulgarisation très général, donne une explication informelle des différents algorithmes et des exemples concrets à foison.

Chapitre 2

La Machine Cellulaire Virtuelle

2.1 Introduction

La MCV (ou Machine Cellulaire Virtuelle) est un langage conçu lors de son D.E.A. [Cor90] et développé dans sa thèse [Cor92] par Thierry CORNU ; elle est complétée et implantée en parallèle sur des transputers par Stéphane VIALLE à l'École Supérieure d'Électricité de Metz.

2.2 Description

La Machine Cellulaire Virtuelle est un modèle à base d'agents[CV93]. Ses agents sont des *cellules* qui s'exécutent « en parallèle ». Le parallélisme massif est simulé, soit intégralement, dans le cas où la machine cible est une machine séquentielle classique, soit partiellement dans le cas où la machine cible est une machine parallèle à transputers.

2.2.1 Le concept de cellule

L'élément de base manipulé par le langage est la *cellule*. Celle-ci est un processeur virtuel comportant des canaux d'entrée et de sortie destinés à être connectés à ceux des cellules voisines, des variables internes et des règles de transition. Un programme en cours de fonctionnement consiste donc en un graphe de cellules en inter-communication. Les actions à effectuer par une cellule sont fixées par un ensemble de règles de transition qui permettent le calcul des nouvelles valeurs de sortie en fonction de celles des entrées.

Le mode de communication entre cellules, est le suivant :

- Les transmissions de données entre les cellules s'effectuent cycliquement : à chaque début de cycle, les informations lisibles dans les canaux d'entrée sont celles qui ont été écrites dans les canaux de sortie correspondant au cycle précédent ou avant ;
- Les données transmises à travers les connexions sont des blocs de 8 octets pouvant représenter une valeur numérique ou un « atome » de 8 caractères, et non des structures de données ou des messages complexes ;
- Il n'y a pas de stockage des messages ou de blocage en écriture ; la transmission est totalement asynchrone. Une nouvelle écriture sur un canal écrase purement et

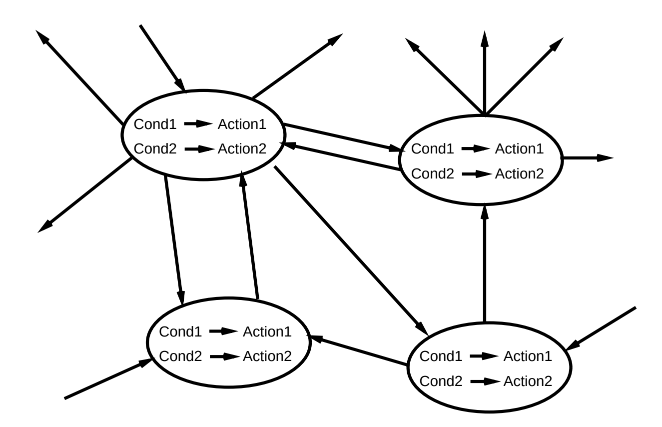


FIG. 2.1 - *Graphe de cellules communiquantes*

simplement la valeur précédemment envoyée, même si elle n'avait pas encore été lue par la cellule réceptrice. Ainsi le mode de communication est de type « partage de mémoire » avec le cycle précédent, où chaque connexion est une unité de mémoire de taille élémentaire accessible en écriture par une cellule exactement et en lecture par un nombre illimité de cellules.

L'idée générale du fonctionnement du système est la suivante : chaque cellule calcule à chaque cycle le nouvel état de ses sorties en fonction de celui de ses entrées courantes.

Toute nouvelle cellule doit être créée en cours de programme, depuis l'intérieur d'une autre cellule, et peut être détruite par la suite. La cellule créatrice sera appelée cellule mère et la cellule résultante sera une cellule fille de la première. Deux cellules ayant la même cellule mère sont appelées des cellules sœurs. Toute cellule d'un programme en cours d'exécution comporte une unique cellule mère et un nombre illimité de cellules filles. La seule exception à cette règle est la cellule *main*, ancêtre de toutes les autres cellules et qui n'est fille d'aucune cellule. Un programme comporte toujours initialement une et une seule cellule (la cellule *main*), à charge pour l'utilisateur de spécifier les caractéristiques de cette cellule.

2.2.2 Types de cellules, structure d'un programme

Chaque cellule du programme est une instance d'un type de cellule qui définit les variables, les références de cellule, les canaux d'entrée, de sortie éventuels de la cellule, ainsi que les règles de transition. Un programme en langage cellulaire consiste en une liste de déclaration de types de cellules. Le langage cellulaire est un langage compilé, c'est-à-dire qu'on ne peut pas créer dynamiquement de nouveaux types ou modifier les types existants. En revanche, il est toujours possible de créer dynamiquement depuis une cellule de nouvelles cellules filles de n'importe quel type déjà connu.

Par ailleurs, les types de cellules sont paramétrables. La valeur des paramètres est fixée au moment de la création dynamique de l'instance, ce qui signifie que différentes cellules de la même classe peuvent avoir des valeurs différentes de leur(s) paramètre(s). Par contre, la nature et le nombre des paramètres sont fixés statiquement pour un type de cellule. Par analogie avec les langages à objets, on peut en gros considérer qu'une déclaration de type de cellule correspond à une déclaration de classe, que la naissance

d'une cellule correspond à une instanciation de classe, et que l'exécution d'une règle correspond plus ou moins à celle d'une méthode.

Les règles de transition déclarées dans une cellule forment un système de règles de production. Chaque remise à jour de cellule comprend deux phases : le test des préconditions de toutes les règles (*matching*) puis le déclenchement d'une règle au plus (*firing*). Les préconditions des règles peuvent être soit des tests sur les valeurs des canaux d'entrée, des paramètres, et des variables internes, soit les tests *initially* et *finally*, qui sont vrais chacun une fois et une seule au cours du programme, respectivement à la création et avant la destruction de la cellule. Des priorités peuvent être attribuées aux règles pour résoudre d'éventuels conflits (le test des prémisses des règles s'effectue dans l'ordre d'écriture,lorsqu'on est dans le même niveau de priorité). Les actions induites par les différentes règles sont des opérations arithmétiques, des créations, connexions et destructions de cellules. Elles sont exprimées dans un langage dont la syntaxe est inspirée des langages procéduraux classiques, tels C et Pascal.

2.3 En pratique

2.3.1 Compilateur existant

Machine cible

Le langage utilisé pour ce travail est un amalgame de la MCV et d'un langage cellulaire, tournant uniquement sur une machine séquentielle : la SPARCStation. Elle tourne sous UNIX. Elle est utilisée par Yannick LALLEMENT qui l'a décrite dans [Lal91]. Il existe une MCV qui tourne sur SUN et Transputers . Le langage cellulaire qui l'utilisera, ParCeL, est en cours de développement.

Compilateur et préprocesseur

Le nom du compilateur est `mcv`, il appelle d'abord le préprocesseur `m4` sur le programme, ce qui implique la possibilité d'inclure des fichiers (`include(nom_du_fichier)`), de créer des macros (`define(nom, définition)`) ou des constantes (de la même manière que les macros).

Les noms des programmes ont en général le suffixe `.m`. Pour compiler le programme, il suffit de taper : `mcv -o pgm pgm.m`. Pour le lancer, taper tout simplement `pgm`.

Lorsqu'une erreur survient, le compilateur donne le numéro de ligne où s'est produite l'erreur : cette ligne correspond au fichier `pgm.mi` qui est le fichier produit par `m4`.

Syntaxe et structure d'un programme

Un programme en langage cellulaire consiste en la déclaration d'une suite de types de cellules. Un type de cellule est une entité syntaxique comprenant :

- une liste de déclarations de variables. Les variables peuvent être de quatre types :
  - variable locale (symbolisée par `0`) ;
  - canal d'entrée (`?`) ;
  - canal de sortie (`!`) ;

  - référence de cellule (`@`).

  On peut, bien entendu, lire et écrire dans une variable locale, uniquement lire dans un canal d'entrée et seulement écrire dans une canal de sortie. Le type référence de cellule est utilisé pour conserver l'immatriculation d'une cellule que l'on crée, en vue de la manipuler dans l'avenir. Toutes les variables peuvent être déclarées sous forme de tableau à une ou plusieurs dimensions de la même manière qu'en C. Le pointeur prédéfini *self* contient, pour toute cellule, la référence à elle-même, et on ne peut s'en servir que dans les instructions `connect` et `born`. Pour la transmettre, on doit utiliser les macros prédéfinies `as_value` et `as_actor`; se référer à [Cor90] pour plus de précision ;

- une liste de paramètres qui peut être déclarée (entre parenthèses, derrière le nom du type de la cellule) ; ces paramètres seront ensuite considérés comme des variables locales qu'on peut lire mais dans lesquelles on ne peut pas écrire ;

- une liste de règles de production définissant le comportement d'une cellule de ce type. Une règle est de la forme :

  ```
  [<coefficient de priorité>][<condition>] ==> <action>
  ```

  À chaque remise à jour de la cellule, les parties `<condition>` des règles sont évaluées. Parmi celles qui sont validées, on choisit la première, ou la plus prioritaire si des priorités ont été assignées aux règles, et on exécute la partie `<action>` correspondante. La partie `<condition>` peut contenir tout type d'expression booléenne, portant éventuellement sur les valeurs des canaux d'entrée de la cellule. Si la partie `<condition>` est absente, la condition sera vraie. La partie `<action>` peut contenir n'importe quelle instruction valide en langage cellulaire. Il existe deux types de règles particuliers : le type *initially* et le type *finally*. La règle dont la prémisse est *initially* est déclenchée lors de la création de la cellule, et celle dont la prémisse est *finally* est déclenchée à la destruction de la cellule, c'est sa dernière action avant d'être éliminée. Ces deux dernières règles sont facultatives.

Les instructions du premier langage cellulaire peuvent être séparées en deux classes :

les instructions classiques : qui s'inspirent plus ou moins directement du langage Pascal. On dispose de

  - *if* :

    ```
    if <condition> then <bloc d'instructions> ;
    ```

    exécute le `<bloc d'instructions>` si la `<condition>` est vraie.

    ```
    if <condition>
       then <bloc d'instructions 1> ;
       else <bloc d'instructions 2> ;
    ```

    exécute le `<bloc d'instructions 1>` si la `<condition>` est vraie et le `<bloc d'instructions 2>` si elle est fausse ;

- *while* :

  ```
  while <condition> do <bloc d'instructions> ;
  ```

  exécute le <bloc d'instructions> tant que la condition est vraie ;
- *for* :

  ```
  for <initial> ;
    to <condition>
    by <itération> ;
    do <bloc d'instructions> ;
  ```

  exécute l'instruction <initial>, puis si la <condition> est fausse, le <bloc d'instructions>, suivi de <itération> et de <bloc d'instructions> tant que la <condition> est fausse [1] ;
- *repeat* :

  ```
  repeat <bloc d'instructions> to <condition> ;
  ```

  exécute le bloc d'instruction tant que la condition est fausse ;
- *les opérateurs et fonctions mathématiques* les plus courantes (+, -, *, /, exp, rand, cos, sin, tan) mais pas (par exemple) le modulo ;
- *les opérateurs de comparaison* sur les expressions numériques rendent des valeurs booléennes. Les opérateurs disponibles sont :

  ```
  <, >, <=, >=, =, <>
  ```

- *les expressions booléennes* `true` et `false` sont les valeurs booléennes de base. *initially* et *finally* sont deux expressions booléennes spéciales. Leur utilisation n'est permise que comme prémisses de règles ;
- *les opérateurs* ***or****,* ***and*** *et* ***not*** permettent d'effectuer de opérations sur les valeurs booléennes.
- *les instructions d'entrée-sortie* qui varient selon les versions de la MCV. Voici celles que j'ai eues à utiliser :
  - *readv*

    ```
    readv(<variable>) ;
    ```

    Met dans la <variable> une valeur numérique lue sur l'entrée standard ;
  - *writev*

    ```
    writev (<variable>) ;
    ```

    Écrit la valeur numérique contenue dans la variable sur la sortie standard ;
  - *newline* Passe à la ligne suivante de la sortie standard ;
  - *writes*

    ```
    writes ("<chaîne>") ;
    ```

1. ATTENTION cette instruction ne fonctionne pas comme celle du C qui s'exécute tant que la condition est vraie.

Écrit le contenu de la chaîne (ou *symbole*) sur la sortie standard. Tous les codes du langage C ne traitant pas de variables sont valables c'est-à-dire que `\t, \n`, ... fonctionnent, mais pas `%s` ni `%d`, par exemple. Il faut mettre les symboles entre doubles quotes ("). Cette instruction ne supporte pas les accents.

les instructions spécifiques il s'agit ici des instructions de manipulation du graphe des cellules. Il y en a trois :

– *born :* création de cellule

```
[<pointeur>] born <type de cellule [(<liste de paramètres>)]> ;
```

Crée une cellule du type donné en lui passant les paramètres, et conserve éventuellement son adresse dans la variable de type pointeur.

– *dead :* destruction de cellule

```
<pointeur> dead ;
```

Élimine la cellule référencée. Par exemple, l'instruction `self dead` tue la cellule où elle apparaît ;

– *connect :* connexion de cellules

```
connect <canal de sortie> in <pointeur 1> oftype <type 1>
     to <canal d'entrée>  in <pointeur 2> oftype <type 2> ;
```

Connecte `<canal de sortie>` de la cellule référencée par `<pointeur 1>` au `<canal d'entrée>` de la cellule référencée par `<pointeur 2>`. Dans le canal de communication ainsi créé ne pourront être transmises que des valeurs numériques ou des atomes (8 caractères au maximum encadrés par des apostrophes).

Il existe un type de cellule particulier : le type main. Par analogie avec le langage C, il doit toujours exister une cellule de type main. Celle-ci est systématiquement créée, en un seul exemplaire, au démarrage de l'exécution du programme. C'est donc dans la cellule main que se trouvent les premières instructions de création de cellules nouvelles. Les nouvelles cellules pourront à leur tour en créer d'autres.

Le langage cellulaire s'exécute en mode synchrone, c'est-à-dire que toutes les cellules existantes sont remises à jour simultanément. Dans les versions précédentes, il existait un mode asynchrone, mais il a été abandonné. Une remise à jour correspond à la sélection et à l'activation d'une règle, c'est une opération insécable, quelle que soit la complexité de la partie `<action>` de la règle sélectionnée.

On dispose d'une liberté supplémentaire concernant la période de remise à jour des cellules, qu'il est possible de modifier. La période par défaut est 1 : la cellule est remise à jour à chaque cycle. Si l'on précise par exemple `period 10` en tête de cellule, elle sera remise à jour une fois tous les 10 cycles seulement.

Enfin, il est possible de faire « dormir» une cellule grâce à l'instruction `<pointeur> sleep;`. Une celllule ainsi endormie n'est plus remise à jour, mais continue d'exister, en particulier ses sorties restent lisibles. Une cellule endormie pourra être « réveillée » grâce à `<pointeur>    awake;`. Ce mécanisme d'inhibition permet d'optimiser les

performances à l'exécution. Yannick LALLEMENT nous signale dans [Lal91] qu'il est également possible d'envisager l'utilisation d'un graphe de cellules endormies comme une structure de données, bien que ce ne soit pas là leur but premier.

Les commentaires Les commentaires sont indispensables dans un programme. Dans le premier langage cellulaire, il en existe de deux sortes : ceux qui sont éliminés par le préprocesseur (`m4`) et qui sont précédés de l'instruction `dnl`, et les commentaires à la manière du C, entourés de `/*` et `*/`. Nous préfèrerons utiliser ceux du C, que `m4` ne supprime pas, car les commentaires insérés dans les macros sont reproduits dans le fichier `.mi`[2].

Il faut noter dans cette implémentation du langage cellulaire le manque de certaines fonctionnalités qui seraient très pratiques, telles les fonctions définissables par l'utilisateur, des fonctions mathématiques, etc. . Ces lacunes seront bientôt comblées par l'arrivée de la nouvelle version du langage cellulaire : ParCeL.

2.3.2 Futur : ParCeL

ParCeL (ou **Par**allel **Ce**llular **L**anguage) est le nom provisoire du deuxième langage cellulaire. Celui-ci est implanté sur la version de la MCV tournant sur machine réellement parallèle et sur machine séquentielle.

Cette version du langage cellulaire sera beaucoup plus proche du C : les instructions à l'intérieur des cellules seront celles du C sans les pointeurs du C, additionnées de celles de manipulation du graphe des cellules, ce qui autorisera l'écriture de fonctions, et non plus seulement de macros comme dans le premier langage, l'utilisation de toutes les fonctions mathématiques du C, ...

De plus, de nouvelles possibilités seront offertes au programmeur en MCV :

- types prédéfinis de cellule pour les entrées-sorties ;
- possibilité de manipuler des fichiers ;
- passage de paramètres à la cellule *main* ;
- instructions de test de connexion (pour savoir si une connexion est ouverte ou non) ;
- possibilité de mettre l'instruction de transmission d'une information dans un canal de sortie avant la déclaration de connexion de ce canal.

---

2. voir page 21

Chapitre 3

Implantation des réseaux de neurones en MCV

3.1 Introduction

Ce chapitre traite de l'implantation d'un type précis de réseaux de neurones en MCV: le perceptron multicouche. La raison de ce choix est simple : c'est le plus couramment utilisé, et il est capable de résoudre des discriminations de complexité quelconque [1]. De plus, il a de nombreuses possibilités à approfondir, comme le problème du dimensionnement des couches cachées.

Son algorithme, comme tous les algorithmes de réseaux de neurones, se parallélise très bien. Les avantages seront la rapidité d'exécution par rapport à une exécution séquentielle, et la possibilité de lancer plusieurs réseaux en même temps.

Nous allons donc voir la démarche utilisée pour l'implantation du perceptron multicouche en MCV et le mode d'emploi de la bibliothèque obtenue.

3.2 Programmation de la bibliothèque `paper`

Nous exposons ici nos choix successifs dans l'écriture de la bibliothèque du **par**allel **per**ceptron : `paper.m`.

3.2.1 Session d'exemple

Le programme `util`, une fois lancé, signale l'architecture du réseau et, à intervalles réguliers, affiche les tests effectués à la fin d'un cycle d'apprentissage. L'intervalle de test est fixé par le nombre de cycles d'apprentissage à la fin desquels les tests ne sont pas effectués. Au début, afin d'avoir un contrôle plus sûr sur les résultats, on les affiche à la fin de chaque cycle d'apprentissage. Le fichier `data.in` contient les vecteurs d'entrée du réseau.

```
anna.parmetie% util < data.in
# Reseau de neurones a retropropagation du gradient
# 3.000000 couches:
# Couche d'entree:                256.000000 neurones
```

1. pour plus de détails sur les avantages de ce type de réseau, se reporter à la section 1.2.2, figure 1.7

```
# Couche cachee 1.000000:       100.000000 neurones
# Couche de sortie:             10.000000  neurones
# 5.000000 cycles 360.000000 exemples 120.000000 tests
# coef 1.000000, Seuil 0.200000, borne 0.500000

Reseau 0.000000 Cycle 1.000000
 Erreur exemples:       0.489373 Reconnaissance:   47.222222 %
 Erreurs inconnus       0.563636 Reconnaissance:   40.833333 %
Reseau 0.000000 Cycle 2.000000
 Erreur exemples:       0.275129 Reconnaissance:   67.500000 %
 Erreurs inconnus       0.374223 Reconnaissance:   55.833333 %
Reseau 0.000000 Cycle 3.000000
 Erreur exemples:       0.117804 Reconnaissance:   86.944444 %
 Erreurs inconnus       0.253440 Reconnaissance:   70.000000 %
Reseau 0.000000 Cycle 4.000000
 Erreur exemples:       0.075000 Reconnaissance:   91.944444 %
 Erreurs inconnus       0.194156 Reconnaissance:   76.666667 %
```

3.2.2 Programme mcv

Il fallait commencer par écrire un programme de perceptron multicouche en MCV que l'on pourrait dans un deuxième temps transformer en une bibliothèque utilisable dans un autre programme. Ceci permettrait de s'assurer de la justesse de l'algorithme effectivement implémenté, en comparant les résultats avec ceux fournis par un programme en C utilisant le même algorithme.

Choix de la granularité

Le problème, au départ, fut de choisir la granularité de notre application, c'est-à-dire s'il fallait voir un réseau de neurones comme une seule cellule, comme autant de cellules que de couches, ou comme une cellule par neurone. Le plus naturel, au vu de l'algorithme (cf. page 12) nous sembla être la dernière solution.

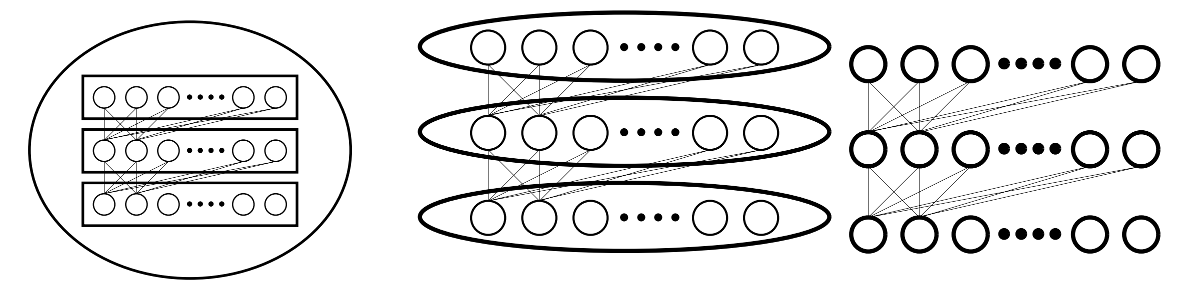


FIG. 3.1 - *Granularité faible : 1 cellule = le réseau*

FIG. 3.2 - *Granularité moyenne : 1 cellule = 1 couche*

FIG. 3.3 - *Granularité fine : 1 cellule = 1 neurone*

Nombre de couches

Pour simplifier le problème, nous avons décidé de démarrer avec un perceptron multicouche constitué de 3 couches. Il nous sera possible (*cf* § *3.2.3*) de généraliser à plus d'une couche cachée puisque les différents types de couches dans un tel réseau sont au

nombre de trois (le type couche d'entrée, le type couche cachée, le type couche de sortie). Les équations étant différentes d'un type de couche à l'autre, nous avons écrit un type de neurone par couche.

Architecture générale

Nous nous sommes tout de suite posé le problème de l'alimentation du réseau : la MCV actuelle n'accepte des données que par l'entrée standard, donc pas question que le programme demande à l'utilisateur le nom d'un fichier de données. Nous avons pris parti pour une cellule qui lit le fichier que l'utilisateur redirige en entrée du programme à son appel : `net <data.in`[2]. La cellule `NeuralNet` crée une cellule `Reader` qui lit les données nécessaires et les distribue au réseau de cellules (*cf. figure 3.4*).

De même, il faut résoudre le problème de l'affichage des résultats. Comme il faut rassembler les sorties de tous les neurones de la couche de sortie, autant créer une cellule `Printer` qui les récupère, effectue les calculs nécessaires et affiche les résultats. Les résultats à afficher sont l'erreur moyenne et le taux de reconnaissance sur les vecteurs appris puis sur les vecteurs inconnus.

Pour une question de simplicité pour l'utilisateur, la cellule appelée `NeuralNet` est chargée de créer toutes les cellules et les connexions du réseau, selon les paramètres fournis. De cette manière, l'utilisateur n'a plus à gérer le réseau, et surtout ses connexions dont la complexité est source d'erreurs. Le lecteur peut se faire une idée de la complexité de ces connexions en jetant un œil à la règle *initially* de la cellule `NeuralNet` de l'annexe A.

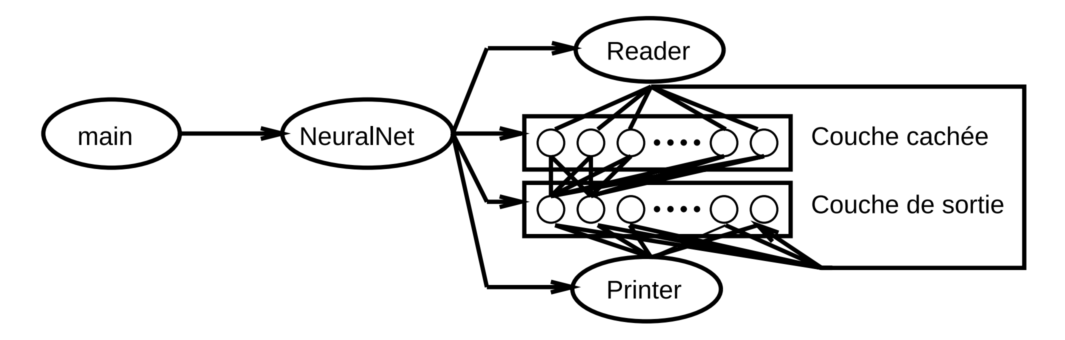


FIG. 3.4 - *L'architecture du réseau*

Synchronisation des cellules

Le plus gros problème restant est celui de la synchronisation des cellules. En effet, chaque cellule étant autonome, il faut qu'elles soient coordonnées. Il est nécessaire que les cellules neurones soient synchronisées pour éviter qu'un neurone d'une couche ne soit en phase de propagation alors qu'un autre de la même couche est en rétro-propagation. Nous avons deux solutions :

- créer des connexions entre une cellule générale `NeuralNet` et les autres cellules. Cette cellule générale donne des ordres par l'intermédiaire de ces connexions. Elles

2. ce fichier fut d'ailleurs converti en ASCII, sous une forme lisible par `readv`, d'un fichier de données venant du simulateur connexionniste SN2

représentent des drapeaux : une connexion avec un neurone pour la propagation, une autre pour la rétro-propagation, une pour signaler que le lecteur doit fournir les données au cycle suivant, etc. ;

- utiliser un compteur de cycle MCV, cycle que nous appellerons *étape* afin de ne pas le confondre avec le cycle d'apprentissage des réseaux de neurones. Tester ce compteur par des règles dans chaque cellule pour synchroniser le réseau sans avoir besoin de connexions supplémentaires.

La première solution a l'avantage d'être plus « naturelle » mais présente l'inconvénient d'ouvrir beaucoup de connexions redondantes et lourdes à gérer. Pour pallier à cet inconvénient, on pourrait utiliser un système de codage simple : à chaque drapeau correspondrait un numéro (puisque les connexions permettent de passer des nombres).

La deuxième solution est se complique petit à petit : le calcul des numéros d'étape est très simple pour le premier vecteur, lors de la phase d'apprentissage. En effet c'est la toute première action du réseau : à la première étape (n° 0), la cellule `reader` propage le vecteur d'entrée vers toutes les cellules de la première couche, ainsi que le vecteur désiré à toutes les cellules de la couche de sortie (type de cellule `Output`). À la deuxième étape (n° 1), les cellules de la première couche cachée effectuent la *propagation* de la couche d'entrée virtuelle vers leur couche : elles calculent la somme de leurs entrées, la seuillent et la mettent dans leurs canaux de sortie. À l'étape suivante (n° 2), la couche suivante (ici, la couche de sortie) fait de même. À l'étape n° 3, le réseau commence la rétro-propagation : elle se fait de la couche de sortie à la couche cachée. L'étape n° 4 est la dernière étape du premier apprentissage du premier vecteur présenté au réseau. Pendant cette étape, la couche cachée rétro-propage l'erreur vers la couche d'entrée (virtuelle) en modifiant les poids des « synapses » qui lui arrivent en entrée.

Les numéros des étapes s'obtiennent directement pour ce premier vecteur, mais dès le deuxième, il faut ajouter le nombre d'étapes nécessaires au traitement d'un vecteur (ici 5). Pour le n[e] vecteur, cela donne :

$$E_n = E_0 + NbEtapesVecteur * n \tag{3.1}$$

avec E_n une étape du traitement du vecteur n° n, et E_0 la même étape pour le vecteur n° 0.

Le même phénomène apparaît pour l'ajout des cycles d'apprentissage, qui reprennent la procédure depuis le début :

$$E_n^c = E_n^0 + NbEtapesCycle * c \tag{3.2}$$

où $NbEtapesCycle$ est le nombre d'étapes nécessaires à tout un cycle, et c le numéro du cycle. Ce qui donne :

$$E_n^c = E_0^0 + NbEtapesVecteur * n + NbEtapesCycle * c \tag{3.3}$$

Le problème se corse lorsqu'on veut ajouter une phase de tests après une phase d'apprentissage : cette phase de tests n'est pas forcément présente après chaque phase d'apprentissage. D'où l'abandon du compteur unique et l'utilisation de drapeaux et de compteurs d'étapes, de vecteur, de cycle. Voici la liste de ces drapeaux et compteurs :

Compteur de vecteur `CV` représente le numéro du vecteur en cours de traitement ;

CE	CV	DAE	DTE	
0	0	1	0	*MyReader* met le vecteur d'exemple dans ses canaux de sortie
1	0	1	0	*Hidden* propagation Reader → Hidden
2	0	1	0	*Output* propagation Hidden → Output
3	0	1	0	*Output* rétro-propagation Output → Hidden
4	0	1	0	*Hidden* rétro-propagation Hidden → Reader
0	1	1	0	idem pour le deuxième vecteur (numéro 1)
⋮	⋮	⋮	⋮	⋮

TAB. 3.1 - *Phase d'apprentissage d'un exemple*

Compteur d'étape `CE` compte les cycles MCV à l'intérieur des phases d'apprentissage ou de tests (remis à zéro à la fin de chaque phase) ;

Compteur de cycle général `CG` comptant le nombre de cycles généraux effectués (c'est-à-dire le nombre de phases d'apprentissage) ;

Drapeau d'apprentissage `DAE` drapeau signalant si on est dans la phase d'apprentissage des exemples ;

Drapeau de test des exemples `DTE` signale si on est dans la phase de test des exemples (c'est-à-dire des vecteurs appris) ;

Drapeau de test des inconnus `DTI` signale si on est dans la phase de test des vecteurs inconnus (non-appris) ;

Drapeau de gestion de l'ensemble `DTG` signale si on est en phase de gestion de cycle d'apprentissage (passage au cycle suivant ou arrêt).

Le tableau 3.1 montre les évolutions des compteurs et des drapeaux pendant les cycles MCV, ainsi que les actions effectuées par les différentes cellules. Le Printer est décalé au départ d'une étape, ce qui implique qu'à son étape n, il peut récupérer les sorties de l'étape n des autres cellules (sorties effectivement effectuées une étape plus tôt). La phase d'apprentissage des exemples (*cf. tableau 3.1*), se répète pour tous les vecteurs d'exemple. Puis la phase de test des exemples (*cf. tableau 3.2*) se répète pour ces mêmes vecteurs. Enfin une partie similaire se déroule pour les vecteurs inconnus ; cette partie est appelée phase de test des inconnus.

3.2.3 Transformation en bibliothèque

À partir de ce programme exemple très fixe (il ne gère que 3 couches), il faut généraliser afin d'offrir à l'utilisateur des facilités de programmation de ces perceptrons multicouches. Cette bibliothèque devant permettre la détermination de la meilleure architecture du perceptron multicouche par rapport à un problème donné. En lançant deux réseaux légèrement différents sur les mêmes données, et en comparant les résultats, on peut déterminer lequel est meilleur. Le tout étant gérable par programme.

CE	CV	DAE	DTE	
0	0	0	1	*MyReader* met le vecteur n° 0 dans ses canaux de sortie
1	0	0	1	*Hidden* propagation Reader → Hidden
2	0	0	1	*Output* propagation Hidden → Output
				MyPrinter propagation Output → MyPrinter
0	1	0	1	idem pour le deuxième vecteur (numéro 1)
⋮	⋮	⋮	⋮	⋮

TAB. 3.2 - *Phase de test des exemples*

Cahier des charges

La bibliothèque

- doit être optimisée en mémoire ;
- doit permettre de ne lire qu'une fois les données nécessaires si elles sont identiques pour plusieurs réseaux ;
- doit gérer des perceptrons multicouches ayant entre 3 et 5 couches ;
- doit permettre le lancement de plusieurs réseaux dans le même programme. Ces réseaux peuvent avoir des paramètres différents ;
- doit permettre le lancement d'un réseau à n'importe quel cycle MCV du programme ;
- doit permettre la consultation des résultats de n'importe quel réseau à n'importe quel moment ;
- doit être simple à utiliser.

Solutions au cahier des charges

L'optimisation de la mémoire n'est que relative à cause de la relative faiblesse du premier langage cellulaire pour la déclaration des tableaux : leur taille doit être fixée dans le texte du programme. La solution retenue utilise `m4` à travers ses constantes pour fixer une limite maximale (tous les tableaux déclarés la prennent en compte) que l'on peut modifier. Voir la section 3.3.5.

La lecture unique des données participe à l'économie de place en mémoire. Elle est gérée par un système de lecteur général. Un lecteur général est une cellule lisant une fois les données et les fournissant à volonté à chaque réseau qui en fait la demande. Pour plus de détails, sur le fonctionnement de cette cellule `GnlReader`, reportez-vous à la section 3.3.2. Cette méthode permettra dans une version future de lancer des réseaux utilisant différentes données. Il n'est pas possible de connaître l'ordre d'exécution des `GnlReader`s, ce qui rend aléatoire l'utilisation de plusieurs de ces cellules avec le langage cellulaire actuel.

Le nombre variable de couches cachées est géré par la cellule `NeuralNet` qui préside à la naissance du réseau. Or la première couche cachée diffère des suivantes (elle n'a pas le même type) car elle a été simplifiée. La première couche cachée n'a pas besoin de diffuser d'information à la couche précédente lors de la rétro-propagation. Nous avons donc supprimé des connexions (et les instructions correspondantes) du type de cellule `Hidden1`.

Comme les connexions sont dépendantes des types de cellules, il a fallu écrire les instructions de connexion des couches séparément pour chaque taille de perceptron multicouche. Par exemple, lorsque le réseau a quatre couches, la première couche cachée est reliée à la deuxième couche cachée, alors qu'avec un réseau a trois couches, la couche cachée est directement connectée à la couche de sortie. Des problèmes similaires se posent entre le perceptron multicouche à 5 couches et celui à 4 couches. C'est pourquoi nous n'avons pas pu intégrer les connexions des couches cachées dans une boucle générale, fonction du nombre de couches cachées. Nous avons donc écrit les trois cas de connexions pour des réseaux de 3 à 5 couches.

Le lancement de plusieurs réseaux dans le même programme est rendu possible par l'élimination des conflits possibles entre deux réseaux : leurs résultats ne s'affichent plus automatiquement sur la sortie standard, il faut le demander explicitement. Chaque réseau a un numéro qui le différencie des autres. Les paramètres de chaque réseau lui sont passés à la naissance. Pour plus de détails, reportez-vous au mode d'emploi de la bibliothèque, page 34.

Lancement d'un réseau à n'importe quel moment dans un programme est rendu possible par le fait que la cellule `GnlReader`, qui *doit* être lancée avant tout réseau, tient à jour une table des réseaux existants et le numéro du vecteur que chaque réseau demande. De plus, chaque réseau est totalement indépendant des cellules qui lui sont extérieures.

La consultation des résultats de n'importe quel réseau peut se faire à n'importe quel cycle MCV du programme, pourvu que la cellule qui veut consulter ces résultats soit la cellule ayant créé le réseau. Ceci se fait par l'intermédiaire des connexions établies entre le `Printer` du réseau et la cellule ayant créé le réseau. Pour les détails, reportez-vous au mode d'emploi, page 37.

La simplicité d'utilisation toute subjective, est permise par l'utilisation de macros `m4`. Malgré les limitations de ce pré-processeur (pas plus de 10 paramètres pour une macro), elles simplifient la vie de l'utilisateur. Par exemple, la macro `DeclareReseaux` écrit automatiquement les déclarations des canaux d'entrées nécessaire à la réception des résultats, la macro `AfficheRes` permet d'éviter de réécrire la fonction d'affichage des résultats. En tout cas, la taille du code nécessaire à écrire pour lancer un réseau est considérablement réduite grâce à l'utilisation du pré-processeur. Voir la section 3.3 et l'annexe A pour comparer la taille des macros à appeler et la place qu'elles occupent réellement.

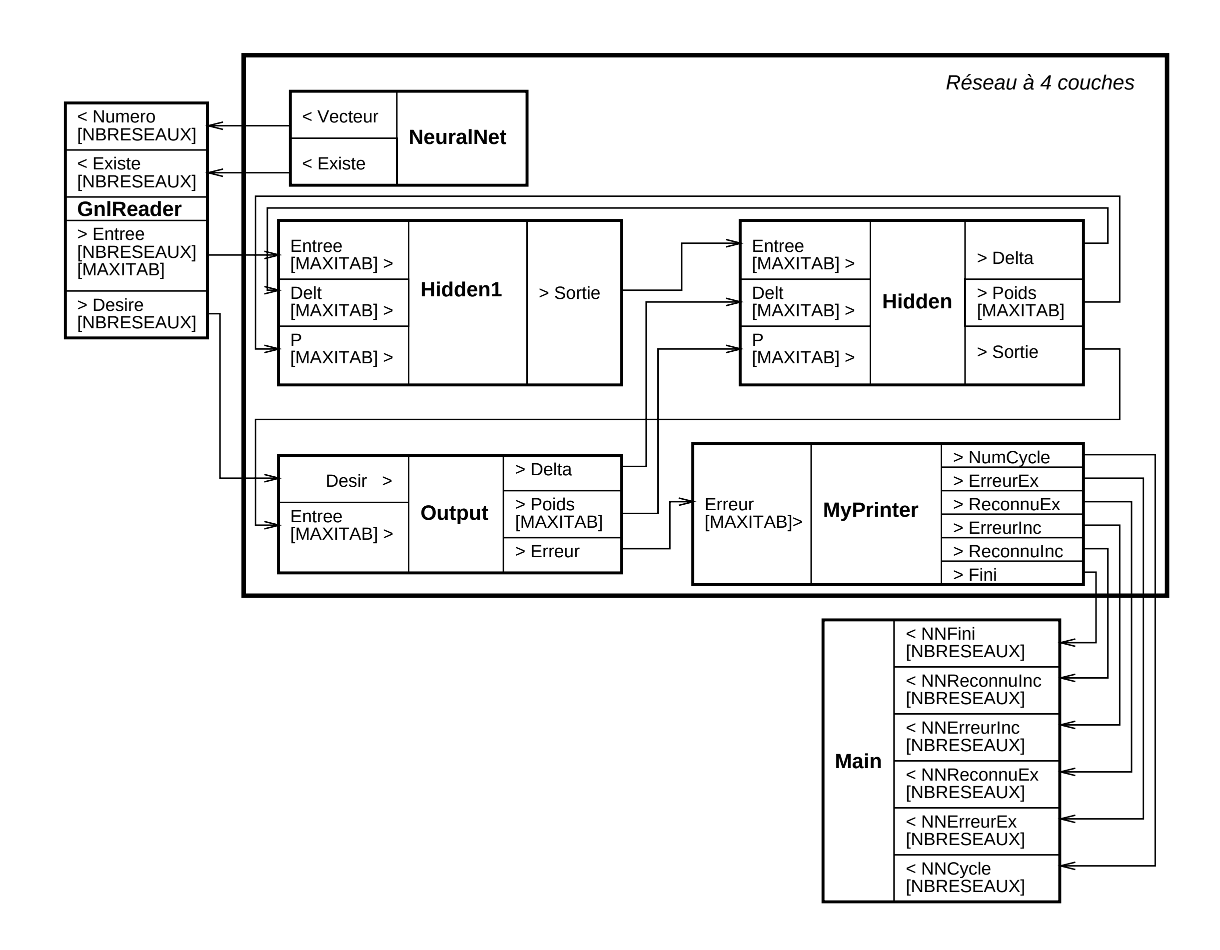


FIG. 3.5 - *Architecture générale d'un réseau à 4 couches*

3.3 Mode d'emploi de la version actuelle

3.3.1 Déclarations

Pour utiliser la bibliothèque `paper.m`, il faut d'abord créer un fichier contenant la cellule *main*, que nous appellerons fichier programme. Il doit impérativement comporter ces instructions :

```
include(paper.m)
actor main {
  DeclareReseaux;
rules
  <les règles de l'acteur main>
};
run;
```

3.3.2 Création d'un réseau de neurones

Pour créer un seul réseau de neurones, il faut ajouter à l'acteur *main* du programme une règle d'initialisation créant l'interface entre les données et le réseau de neurones, et

demandant la naissance et le démarrage du réseau de neurones.

Il y a différents paramètres à passer à la bibliothèque par l'intermédiaire des deux instructions `born GnlReader (...);` et `born NeuralNet (...);` :

born GnlReader cette instruction n'a que deux paramètres : la taille du vecteur d'entrée du réseau et la taille de sa couche de sortie. Mais il est indispensable de retenir l'*adresse* du lecteur des données, afin que le réseau (ou les réseaux, voir la partie 3.3.4) puisse être relié au lecteur. Il faut donc fournir au réseau un pointeur de cellule : dans le cas normal d'un seul format de données, cette immatriculation sera `GnlR`, qui est déclarée par l'intermédiaire de la macro `DeclareReseaux`. Syntaxe de cette instruction :

```
GnlR born GnlReader(<taille d'entrée>, <taille de sortie>);
```

born NeuralNet cette instruction a 15 paramètres que nous allons détailler :

1° **nombre de couches** du réseau en comptant la couche d'entrée, la couche de sortie et les couches cachées ;

2° **taille d'entrée** du réseau ou taille de sa couche d'entrée virtuelle[3] ;

3° **taille de la première couche cachée** du réseau, elle doit forcément être supérieure à zéro car le réseau a au moins trois couches ;

4° **taille de la deuxième couche cachée** du réseau. Si celui-ci a moins de 4 couches, mettez ce paramètre à zéro ;

5° **taille de la troisième couche cachée** du réseau. Si celui-ci a moins de 5 couches, mettez ce paramètre à zéro ;

6° **taille de la couche de sortie** du réseau. Chaque neurone de sortie correspond à une classe à discriminer. Par exemple, dans la reconnaissance de chiffres, il y aurait 10 neurones de sortie ;

7° **Nombre de cycles** d'apprentissage du réseau ;

8° **Nombre d'exemples** à apprendre à chaque cycle. ATTENTION le nombre d'exemples plus le nombre de vecteurs inconnus doit rester inférieur au nombre total de vecteurs dans le fichier des données, sinon, le programme vous signale que les resultats fournis risquent d'être erronés ;

9° **Nombre d'inconnus** : nombre de vecteurs à tester pour vérifier le comportement du réseau sur des vecteurs non appris. Même remarque que cidessus ;

10° **Seuil d'erreur** : seuil d'erreur quadratique au-dessus duquel un résultat de propagation est considéré comme étant un échec (le vecteur d'entrée n'a pas été «reconnu» avec une assez grande précision) ;

11° η **: coefficient d'apprentissage** qui intervient dans la rétro-propagation du gradient (voir l'algorithme page 12) ;

12° **Borne** d'initialisation des poids des neurones. Les poids des connexions entre les neurones sont initialisés aléatoirement entre $-borne$ et $+borne$. Il est conseillé d'utiliser une petite valeur décimale (de l'ordre de 1);

3. cette couche n'existe pas dans l'implantation, voir la section 3.2.2

13° **Numéro** du réseau. ATTENTION ce numéro doit être *unique*, strictement inférieur à la constante **NBRESEAUX**[4] et avoir une valeur entière. Il sera utilisé lors de la récupération des résultats. Si plusieurs réseaux ont le même numéro, les résultats récupérés ne seront pas fiables ;

14° **Cellule appelante** : pointeur sur la cellule d'où est appelé le réseau (ou cellule mère du réseau), ce ne peut être que la cellule *main*. ATTENTION, le pointeur doit être transformé en valeur numérique par l'intermédiaire de la macro `as_value`. En général, on utilisera la macro définie par la bibliothèque : `Self`[5] ;

15° **GnlReader** : pointeur sur le lecteur général, même remarque qu'au-dessus, la bibliothèque définit une macro `GR` qui est `as_value(GnlR)`.

La syntaxe de cette instruction est donc :

```
born NeuralNet(<nb de couches>,
               <taille d'entrée>, <taille couche cachée 1>,
               <taille couche cachée 2>, <taille couche cachée 3>,
               <taille de sortie>,
               <nb de cycles>,
               <nb d'exemples>, <nb d'inconnus>,
               <seuil d'erreur>, <coefficient d'apprentissage>,
               <borne d'initialisation>,
               <numéro du réseau>, <cellule appelante>, <GnlReader>);
```

Voici un exemple de lancement d'un réseau à 3 couches, avec une couche d'entrée de $16*16$ bits, c'est-à-dire de 256 neurones, une couche cachée de 100 neurones, la couche de sortie ayant 10 neurones. Nous reprenons toujours notre exemple de reconnaissance de chiffres avec 25 cycles d'apprentissage, 360 exemples à apprendre et 120 inconnus pour le test final des performances du réseau. En général, on prend $2/3$ des vecteurs disponibles pour l'apprentissage et $1/3$ pour le test. Nous utiliserons un seuil d'erreur de 0,2 avec un coefficient d'apprentissage à 1, et le numéro du réseau à zéro. *On commence toujours par le numéro zéro* :

```
/* ****************************************************
   essai.m : Programme d'essai de la bibliothèque paper.m
   ****************************************************
   Version date    Commentaires
   00      28/6/93
                   Premier essai : on lance un reseau sans
                   recuperer les resultats.
   ****************************************************/

include(paper.m)

/* ====================================================
   =============== Cellule Principale
   ==================================================== */
actor main{
```

4. *cf* § *3.3.5*, par défaut : 2

5. notez la majuscule

```
  /* Declaration des canaux d'entree pour la reception des
     resultats du reseau. Declaration de GnlR                */
  DeclareReseaux;

  rules

    /* ***** Requete de generation et de lancement du reseau */
    initially==>{

      /* Requete de generation et de lancement du lecteur    */
      GnlR born GnlReader(256,10);

      /* Requete de generation et de lancement du reseau     */
      born
      NeuralNet(3,256,100,0,0,10,25,360,120,0.2,1.0,0.5,0,Self,GR);

    }; /* *****                                              */
};

/* indispensable pour que le programme demarre               */
run;
```

3.3.3 Récupération des résultats finaux

Pour que la bibliothèque soit utile, il faut pouvoir récupérer les résultats des calculs d'un réseau. C'est pourquoi nous allons utiliser deux drapeaux : un nous signalant si le réseau a fini ses calculs (`NNFini[<numéro du réseau>]`), et un autre pour savoir si nous avons déjà utilisé ses résultats (`NNVu[<numéro du réseau>]`).

Le drapeau `NNFini` est un canal d'entrée venant directement du réseau, nous ne pourrons donc pas écrire dedans. Par contre, le drapeau `NNVu` est une variable de la cellule appelante, c'est pourquoi il faut l'initialiser avant que le réseau ne commence à s'exécuter. C'est donc dans la règle *initially*, si c'est là qu'on demande le lancement du réseau, qu'on initialise `NNVu` grâce à une macro appelée `InitVus`. Incidemment, si nous voulions déclarer une variable `NNijkz`, nous aurions une erreur à la compilation car cette variable serait déjà déclarée (c'est un compteur de boucle utilisé par la macro).

Une fois que nous savons que le réseau a fini (`NNFini[<numéro du réseau>]=TRUE`) et que nous n'avons pas encore exploité les résultats (`NNVu[<numéro du réseau>]=FALSE`), nous pouvons les afficher.

La macro `DeclareReseaux` a déclaré des canaux d'entrées contenant tous les résultats souhaitables :

`NNCycle[<numéro du réseau>]` qui contient à tout moment du calcul le numéro du cycle d'apprentissage en cours. Permet de savoir où en est l'apprentissage ;

`NNErreurEx[<numéro du réseau>]` qui contient l'erreur moyenne sur tous les exemples appris ;

`NNReconnuEx[<numéro du réseau>]` qui contient le pourcentage de reconnaissance sur les exemples appris ;

`NNErreurInc[<numéro du réseau>]` qui contient l'erreur moyenne sur les exemples inconnus ;

`NNReconnuInc[<numéro du réseau>]` qui contient le pourcentage de reconnaissance sur les exemples inconnus.

Ces canaux sont lisibles à tout moment mais leur contenu n'est pertinent qu'à la fin d'une phase de test des inconnus, qui se produit par défaut uniquement au dernier cycle [6], lorsque `NNFini[<numéro du réseau>]` est vrai. On peut faire des calculs sur les résultats fournis.

Afin d'afficher les résultats, la bibliothèque fournit une macro `AfficheRes(<numéro du réseau>)`. Cette macro met le drapeau `NNVu[<numéro du réseau>]` à vrai lors de l'affichage du dernier cycle. Une autre macro, simplifiant la frappe, est `Fin(<numéro du réseau>)` qui remplace `(NNFini[<num>]=TRUE and NNVu[<num>]=FALSE)`.

Reprenons notre exemple pour lui faire afficher les résultats :

```
/* ****************************************************
   essai.m : Programme d'essai de la bibliothèque paper.m
   ****************************************************
   Version date    Commentaires
   01      29/6/93
                   Deuxieme essai : on lance un reseau et
                   on affiche les resultats.
   ****************************************************/

include(paper.m)

/* =====================================================
   =============== Cellule Principale
   ===================================================== */
actor main{

  DeclareReseaux;

  rules

    /* ***** Requete de generation et de lancement du reseau */
    initially==>{
      /* Requete de generation et de lancement du lecteur    */
      GnlR born GnlReader(256,10);

      /* Requete de generation et de lancement du reseau     */
      born
      NeuralNet(3,256,100,0,0,10,25,360,120,0.2,1.0,0.5,0,Self,GR);

      InitVus;
    }; /* *****                                              */

    /* ***** Fin du calcul et affichage des resultats        */
    Fin(0) ==> {
```

6. voir la section 3.3.5 pour plus de détails

```
      AfficheRes(0);
    }; /* *****                                                 */

}; /* ===============                                           */

/* indispensable pour que le programme demarre                  */
run;
```

Pour compiler ce programme (voir la page 21), tapez `comp -o essai essai.m`. Pour le lancer, tapez `essai`. Ne vous étonnez pas de voir plusieurs fois *exactement* le même résultat : pour un même programme, la fonction aléatoire fournit toujours la même suite de valeurs. Si vous lancez plusieurs fois de suite exactement le même programme, les résultats seront identiques.

3.3.4 Création de plusieurs réseaux

Pour créer (et lancer) un deuxième réseau, il suffit d'ajouter une ligne : l'instruction `born NeuralNet`[7]. Les deux réseaux s'exécuteront alors en parallèle, complètement indépendamment l'un de l'autre. Pour récupérer les résultats de ce deuxième réseau, il suffit d'ajouter une règle qui se déclenchera lorsqu'il aura fini. En général, on ne peut prévoir l'ordre de fin d'exécution des réseaux. Le raisonnement sur les résultats ne se basera donc pas sur cet ordre.

```
/* *****************************************************
   essai.m : Programme d'essai de la bibliothèque paper.m
   *****************************************************
   Version date    Commentaires
   02      29/6/93
                   Troisième essai : on lance un deuxieme
                   reseau et on affiche aussi ses resultats.
   *****************************************************/

include(paper.m)

/* =====================================================
   =============== Cellule Principale
   ===================================================== */
actor main{

  DeclareReseaux;

  rules

    /* ***** Requete de generation et de lancement du reseau      */
    initially==>{
      /* Requete de generation et de lancement du lecteur         */
      GnlR born GnlReader(256,10);

      /* Requete de generation et de lancement du premier reseau  */
      born
```

7. Attention : la configuration par défaut prévoit la création et le lancement de deux réseaux au maximum, pour savoir changer ce nombre, voyez la section 3.3.5

```
      NeuralNet(3,256,100,0,0,10,25,360,120,0.2,1.0,0.5,0,Self,GR);

      /* Requete de generation et de lancement du deuxieme reseau */
      born
      NeuralNet(4,256,100,50,0,10,25,360,120,0.2,1.0,0.5,1,Self,GR);

      InitVus;
    }; /* *****                                                  */

    /* ***** Fin du calcul et affichage des resultats du premier
             reseau                                              */
    Fin(0) ==> {
      AfficheRes(0);
    }; /* *****                                                  */

    /* ***** idem pour le deuxieme reseau                        */
    Fin(1) ==>{
      AfficheRes(1);
    }; /* *****                                                  */

}; /* ==============                                             */

/* indispensable pour que le programme demarre                   */
run;
```

Le nombre de réseaux exécutables en parallèle n'est limité que par les capacités de la machine sur laquelle on travaille.

3.3.5 Constantes

Les constantes définies dans la bibliothèque `paper.m` peuvent être redéfinies par l'utilisateur. Pour la plupart, elles pallient un défaut du compilateur actuel de la MCV : on ne peut pas déclarer de tableaux dynamiques, donc il nous faut une taille maximale pour certains tableaux. Voici les constantes redéfinissables :

NBRESEAUX définit le nombre de réseaux qu'on peut gérer simultanément. Par défaut, elle est à **2**. Si vous projetez de n'utiliser qu'un seul réseau et que vous voulez gagner de la place en mémoire, mettez cette constante à 1. Par contre, vous pouvez utiliser plus de 2 réseaux simultanément ;

MAXITAB définit la grandeur des tableaux en général. Comme dans nos essais nous n'avons pas utilisé de couche de neurones plus grande que la couche d'entrée, nous l'avons fixée par défaut à la taille de la couche d'entrée : **256**. Si vous utilisez un ou plusieurs réseau de neurones dont toutes les couches, y compris la couche d'entrée, sont plus petites que cette valeur par défaut, vous devez y mettre la taille de votre plus grosse couche ;

NBVECTEURS définit le nombre maximal de vecteurs dans le fichier d'entrée (exemples + inconnus). Par défaut, la valeur est de **480**, qui correspond au nombre de vecteurs dans le fichier `data.in`. Cette valeur doit être exacte ou inférieure aux nombre de vecteurs du fichier, sous peine de non-terminaison de l'exécution du programme ;

NBRAPIDES définit le nombre de cycles dits « rapides » durant lesquels le réseau ne fait pas de test (ni sur les inconnus ni sur les exemples) avant de faire un cycle qui en fait. Par défaut, la valeur est égale au nombre de cycles à faire moins un, afin que les tests soient effectués uniquement au dernier cycle. Par exemple, pour afficher les résultats à chaque cycle, mettez **NBRAPIDES** à zéro.

Pour changer la valeur d'une de ces constantes, redéfinissez-la simplement en tapant (avant le `include(paper.m)`) : `define(<CONSTANTE>,<valeur>)`. Un exemple complet de programme utilisant la bibliothèque est présenté dans l'annexe B, page 83.

3.3.6 Récupération des résultats intermédiaires

On peut vouloir surveiller l'évolution du réseau de neurones pendant son apprentissage, ou récupérer ses résultats pour étudier son comportement. Il nous faut pour cela utiliser la constante **NBRAPIDES** (*cf.* § *3.3.5*).

La bibliothèque `paper.m` met à votre disposition une macro *à ne pas utiliser séparément de* `AfficheRes` : `Teste`. L'argument de cette macro est le numéro du réseau à tester. Elle teste si le réseau en est à un cycle où les tests ont été effectués et diffusés dans les canaux d'entrées déclarés par `DeclareReseaux` (*cf.* § *3.3.3*). La macro `AfficheRes` maintient à jour la variable `NNCC` qui contient le dernier cycle où les tests ont été effectués.

L'utilisation de cette macro s'insère très bien dans la version 01 du programme `essai.m`, il suffit d'ajouter cette règle dans la cellule *main* :

```
Teste(0)==>{
  AfficheRes(0);
};
```

3.4 Comparaisons bibliographiques

De nombreuses implantations du perceptron multicouche ont déjà été faites sur des machines parallèles. C'est l'objet de cette comparaison qui portera plus sur l'algorithmique que sur les performances que nous ne saurions comparer.

3.4.1 Implantation en langage cellulaire

Machine Cellulaire Virtuelle définition, implantation et exploitation [Cor92]

Thierry CORNU a implanté un algorithme de rétro-propagation du gradient dans le cadre de sa thèse.

L'architecture du perceptron multicouche programmé par Thierry CORNU est assez différente de celle que nous avons choisie : il utilise un type de cellule `synapse` qui n'apparaît pas chez nous. De plus, une cellule `séquenceur` s'occupe de la synchronisation de toutes les cellules. Nous n'avons pas opté pour cette méthode pour des questions d'occupation de mémoire (*cf. page 29*).

De plus l'architecture choisie fonctionne avec des cellules de type `couche` qui s'occupent de la création des neurones d'une couche et des connexions avec les neurones des autres couches. Ceci est possible grâce à une utilisation fine des possibilités de `m4`.

Mais ce programme n'est pas prévu pour un corpus normal. Le corpus utilisé n'est composé que de 4 vecteurs de 2 composants. Le temps nécessaire à une propagation est fixé arbitrairement et approximativement à 10 cycles MCV, ce qui est assez lent pour un réseau constitué de 2 couches.

Le programme de Thierry CORNU est donc une validation du fait qu'on peut programmer un perceptron multicouche en MCV, mais pas vraiment un programme d'exploitation pour un problème réel.

3.4.2 Répartition colonne

Contrairement à nous, les auteurs d'implantations de quelque algorithme que ce soit sur une machine parallèle doivent prendre en compte la topologie du réseau de processeurs sur lequel ils travaillent. Les auteurs des articles ci-dessous ont opté pour une répartition particulière du perceptron multicouche sur leurs processeurs (*cf. figure 3.6*). Sur un système à p processeurs, chaque couche du réseau est divisée en p sous-ensemble. Chaque sous-ensemble est affecté à un processeur.

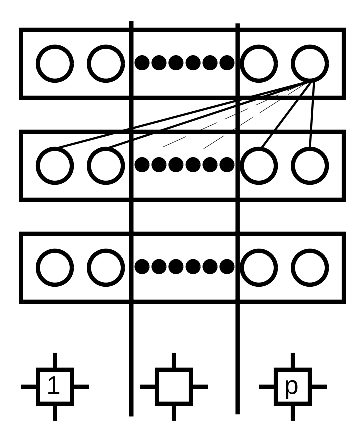


FIG. 3.6 - *Répartition en colonnes du perceptron multicouche sur p processeurs*

Performance of backpropagation on a parallel transputer-based machine [Ern88] Christine ERNOULT utilise la répartition colonne du perceptron multicouche sur un réseau de 16 transputers T800. La mémoire de chaque processeur étant de 1 méga-octet, elle stocke la table de répartition des neurones sur le réseau de transputers. À chaque neurone sont attachés les poids des liaisons avec les neurones de la couche suivante ;

Multilayer neural networks on distributed-memory processors [YN90] Un problème différent de la répartition des neurones sur les processeurs a dû être résolu : les communications entre les processeurs. En effet, chaque transputer n'a que 4 canaux pour communiquer avec les autres processeurs. Chaque neurone est susceptible de communiquer des informations avec tous les neurones d'une autre couche qui peuvent se trouver répartis sur plus de 4 transputers. Deux modes de communications sont disponibles: la *diffusion* et la *communication personnalisée*. La

diffusion est utilisée lorsque les données transmises doivent parvenir à tous les autres processeurs (par exemple pour communiquer l'état des neurones). Lorsque les données ne doivent parvenir qu'à un seul processeur, il faut utiliser la communication personnalisée. C'est le cas lors de la communication de tableaux de calculs partiels nécessaires au calcul des coefficients d'erreur ;

Implantation de l'algorithme de rétro-propagation du gradient sur une machine hypercube [WR91] Le problème principal (excepté la distribution du réseau sur les différents processeurs) est encore la réduction du temps de communication. Wang et Robert essayent d'exploiter la capacité de communication de l'hypercube au maximum en utilisant un algorithme d'accumulation. C'est-à-dire que les informations sont rassemblées et condensées avant d'être envoyées, ce qui réduit la taille et le nombre des communications ;

Parallélisation de l'algorithme de rétro-propagation du gradient récursif [Fry91] Frydlender considère plus le problème sous son aspect matriciel. Il utilise un découpage sensiblement identique du réseau mais en distinguant poids synaptiques et neurones.

La différence avec la MCV tient principalement aux faits que nous ne nous préoccupons pas de savoir sur quel processeur implanter tel ou tel neurone, et que les communications entre les neurones sont identiques, au moins en apparence pour le programmeur, quel que soit le neurone.

3.4.3 Distribution des données

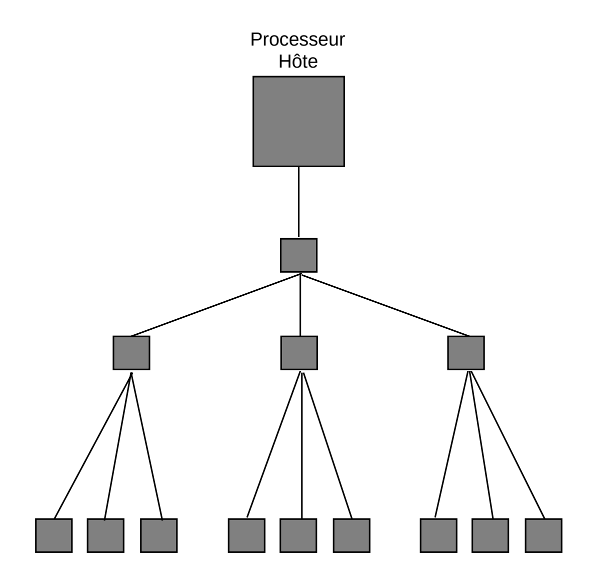


Fig. 3.7 - *Topologie pyramidale*

Étude d'un réseau de neurones multi-couches pour l'analyse du sommeil sur T-Node [PST+90] les auteurs ont opté pour une autre répartition du réseau sur

les différents processeurs. La topologie de leur réseau de transputers est pyramidale (*cf. figure 3.7*).

Pour le cas particulier d'un petit perceptron multicouche à 3 couches, chaque processeur contient une partie du réseau et distribue les données qu'il n'utilise pas à 3 processeurs fils. Cette technique est appelée *parallélisme réplicatif (ou vectoriel)* : chaque processeur exécute le même algorithme sur des données différentes (**S**ingle **P**rogram **M**ultiple **D**ata). Leur algorithme de rétro-propagation utilise un moyennage des erreurs des processeurs « fils » avant la modification des poids du perceptron multicouche.

Pour un corpus de données de petite taille, cette méthode assure, selon les auteurs, d'excellentes performances ;

On a parallel algorithm for back-propagation by partitioning the training set [PM92] L'implantation a ici été effectuée sur un anneau de transputers T800. Une copie du même perceptron multicouche est implémentée sur chaque processeur. Tous les processeurs reçoivent un paquet d'exemples différents et les traitent séparément. La seule communication nécessaire intervient lors de la mise à jour des poids. Le problème traité par Hélène PAUGAM-MOISY tient au fait que plus les paquets sont grands, plus la vitesse de traitement est grande (moins de communications), et qu'inversement, plus les paquets sont petits, plus la convergence du réseau est rapidement atteinte. Elle cherche donc le meilleur compromis entre l'algorithme du gradient stochastique et celui du gradient total.

Trois lois expérimentales sont retirées.

Elle conclut qu'il est possible d'obtenir un *speedup* pour la rétro-propagation par la distribution de l'ensemble des exemples, mais qu'il faut être prudent lors du choix du nombre de processeurs, et de la taille des partitions des exemples.

Comme `paper.m` est implanté en MCV et que le programmeur ne peut pas savoir sur quel processeur tournera une cellule, nous n'avons pas utilisé la technique de la distribution des données. Nous aurions toutefois pu utiliser le principe en utilisant une cellule comme un processeur mais cette implantation n'aurait pas été aussi efficace que celles exposées dans les articles précédents. En effet une cellule n'aurait pas correspondu exactement à un processeur et les calculs d'optimisation n'auraient pas été pertinents. Il aurait fallu pouvoir répartir précisément certaines instances de cellules sur les processeurs disponibles.

3.4.4 Pipeline

Network learning on the Connection Machine [BR87] BLELLOCH et ROSENBERG ont utilisé une Connection Machine. L'originalité de ce travail réside dans le « pipelinage » de l'algorithme : pour éviter de laisser des processeurs inactifs, on propage un vecteur à travers une couche tandis que le vecteur précédent traverse la couche suivante. Les auteurs ont testé leurs idées avec NETtalk.

Cette démarche intéressante pour la rapidité d'exécution des programmes ne l'était pas pour la clarté d'écriture de notre programme, c'est pourquoi nous ne l'avons pas retenue. Elle pourra être développée à l'avenir.

3.5 Conclusion

3.5.1 Méthodes de synchronisation

La synchronisation des cellules est un des problèmes principaux ([Cor92, section IV.1.2-3]) en programmation cellulaire. Nous avons vu plus haut (*cf.* § *3.2.2, page 29*) deux méthodes principales, chacune ayant ses avantages et ses inconvénients :

– l'utilisation d'une cellule séquenceur, connectée à toutes les cellules à synchroniser, et qui leur signifie au moment adéquat l'action à effectuer. Cette solution est relativement coûteuse en nombre de connexions ;

– l'utilisation d'une « horloge » locale à chaque cellule qu'il est nécessaire de synchroniser avec les autres. Il est alors indispensable de démarrer toutes les cellules à synchroniser lors du même cycle MCV (à moins de mettre en place un système de passage de décalage par les paramètres).

3.5.2 Limites de la version réalisée

Les limites de cette version sont :

– sa *lenteur* (par rapport aux algorithmes séquentiels optimisés de simulateurs existants, mais surtout par rapport aux implantations parallèles des perceptrons multicouches [Ern88, WR91, PST+90, etc.]). C'est principalement dû au fait que le programme est prévu pour fonctionner en parallèle et qu'il tourne sur une machine séquentielle ;

– l'impossibilité de *sauvegarde des poids* des réseaux obtenus (la seule manière de le faire serait de les afficher dans la sortie standard). Mais cela pourra se faire en ParCeL.

3.5.3 Idées d'amélioration de `paper`

Cette version de la bibliothèque peut encore être améliorée :

– on peut « pipeliner » la propagation, puis la rétro-propagation, en s'inspirant des travaux exposés dans [BR87]. Cet apport devrait permettre d'augmenter notablement les performances ;

– l'apport de plus de souplesse dans l'architecture des réseaux utilisés serait appréciable. En effet, `paper.m` n'est prévu que pour des réseaux entièrement connectés, mais il peut arriver qu'on veuille connecter une partie d'une couche avec seulement une partie d'une autre couche. Cette méthode permet, dans des cas très particulier où l'on peut souhaiter cette séparation d'une couche en plusieurs parties, d'économiser un nombre de connexions non négligeable. Elle permettrait ainsi d'éviter des calculs inutiles. Cette modification au programme n'est toutefois pas indispensable, et ne serait pas forcément utilisée souvent ;

– il faut adapter le `GnlReader` à des données autres que celles utilisées pour le test de `paper.m`: jusqu'à présent, les sorties désirées sont uniquement calculées par la cellule de lecture, alors qu'il faudrait maintenant qu'elle puisse les lire. Il faut soit

modifier le format des données en entrée en juxtaposant les sorties désirées aux vecteurs d'entrée, soit lire les vecteurs désirés après tous les vecteurs d'entrée (ce qui implique de connaître le nombre de vecteurs dans le fichier d'entrée) ;

- on pourrait ajouter la possibilité de sauvegarder les poids du réseau, pour pouvoir les réutiliser dans un autre programme ou continuer un apprentissage ;
- la possibilité de donner une liste de numéros de cycles d'apprentissage pour lesquels on veut que les tests soient effectués, ce qui permettrait, par exemple, de tester la convergence sur les derniers cycles prévus.

Conclusion

L'écriture de cette bibliothèque nous a confronté à un problème algorithmique : la synchronisation de cellules indépendantes faisant partie d'un ensemble devant être synchronisé. Nous avons résolu ce problème après exploration des diverses solutions que nous avons développées.

La prochaine étape devrait être l'étude du meilleur nombre de réseaux à lancer simultanément pour que le gain de temps reste significatif par rapport à une exécution en séquence de ces réseaux. Il devrait en effet exister une limite au-delà de laquelle la gestion de nombreuses cellules en MCV prend plus de temps que l'exécution de leurs instructions.

Bibliographie

[BR87] Blelloch (Guy) et Rosenberg (Charles R.). – Network learning on the connection machine. *?*, 1987(?), pp. 323–326.

[Coc88] Cochet (Yves). – *Réseaux de neurones.* – résumé de cours DESS-ISA option IA n 389, IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires), Janvier 1988.

[Cor90] Cornu (Thierry). – *Langage cellulaire (version expérimentale, manuel réduit d'utilisation).* – Rapport interne n 2 bis, Service Intelligence Artificielle et Productique, Metz, École Supérieure d'Électricité, Mai 1990. révisé Janvier 1991.

[Cor92] Cornu (Thierry). – *Machine Cellulaire Virtuelle définition, implantation et exploitation.* – Thèse, Université de Nancy I, Octobre 1992.

[CV93] Cornu (Thierry) et Vialle (Stéphane). – A framework for implementing highly parallel applications on distributed memory architectures. *Leeds*, 1993, p. 24.

[Ern88] Ernoult (Christine). – Performance of backpropagation on a parallel transputer-based machine. *In : Neuro-Nîmes'88*, pp. 311–324.

[Fry91] Frydlender (Hervé). – Parallélisation de l'algorithme de rétropropagation du gradient récursif. *In : Neuro-Nîmes'91*, pp. 639–652.

[Hat89] Haton (Jean-Paul). – Modèles connexionnistes pour l'intelligence artificielle, 1989(?).

[Lal91] Lallement (Yannick). – *Machine Cellulaire Virtuelle et Intelligence Artificielle.* – Rapport technique, Supélec Metz, IAP, Septembre 1991. Rapport de D.E.A.

[Lhe91] Lhermitte (Claude). – *Intelligence Artificielle et Connexionnisme.* – Rapport technique n 10860/..a1991, Supélec, 1991. 11 pages.

[Lip87] Lippmann (Richard P.). – An introduction to computing with neural nets. *IEEE ASSP Magazine*, Apr 1987, pp. 4–22.

[PDG88] Personnaz (Léon), Dreyfus (Gérard) et Guyon (Isabelle). – Les machines neuronales. *La Recherche*, vol. 19, n 204, Novembre 1988, pp. 1362–1371.

[PM92] Paugam-Moisy (Hélène). – On a parallel algorithm for back-propagation by partitioning the training set. *In : Neuro-Nîmes'92*, pp. 53–65.

[PST+90] Pinti (A.), Schaltenbrand (N.), Toussaint (M.), Gresser (J.), Luthringer (R.), Minot (R.) et Macher (J.P.). – Étude d'un réseau de neurones multi-couches pour l'analyse du sommeil sur t-node. *La lettre du transputer et des calculateurs distribués*, décembre 1990, pp. 21–32.

[WR91] Wang (Shengrui) et Robert (François). – Implantation de l'algorithme de rétro-propagation du gradient sur une machine hypercube. *Technique et Science Informatiques*, vol. 10, n4, 1991, pp. 297–303.

[YN90] Yoon (Hyunsoo) et Nang (Jong H.). – Multilayer neural networks on distributed-memory multiprocessors. *In: International Neural Network Conference*, pp. 669–672. – Palais des Congrès – Paris – France, juillet 1990.

Index

Annexe A

Bibliothèque paper.m

```
/* ***************************************************************************
   paper.m : Programme de reseau de neurones a retropropagation du gradient.
   ***************************************************************************
   Revision  date   Commentaires
   18       16/7/93

                    Changement des noms des variables dans les macros:
                    Cycle[] devient NNCycle, Fini devient NNFini, Vu
                    devient NNVu, CC devient NNCC, ErreurEx devient
                    NNErreurEx, ReconnuEx devient NNReconnuEx,
                    ErreurInc devient NNErreurInc, ReconnuInc devient
                    NNReconnuInc.

   ***************************************************************************/

define(PROGRAMME,paper.m)
define(REVISION,18)

/* ----- Macros et constantes independantes des parametres d'un reseau ----- */

/* Nombre de cycles rapides durant lesquels on ne fait pas de test (ni
   sur sur les inconnus ni sur les exemples) avant de faire un cycle
   qui en fait */

ifdef(`NBRAPIDES',,
`define(NBRAPIDES,(NbCycles-1))'
)

/* Nombre maximal de reseaux a gerer simultanement */
ifdef(`NBRESEAUX',,
`define(NBRESEAUX,2)'
)

/* Nombre maximal de neurones dans une couche = taille maximale des tableaux */
ifdef(`MAXITAB',,
`define(MAXITAB,256)'
)
```

```
/* Nombre maximal de vecteurs dans le fichier d'entree (exemples+inconnus) */
ifdef(`NBVECTEURS',,
`define(NBVECTEURS,480)'
)

/* Valeur du booleen Vrai */
define(TRUE,1)

/* Valeur du booleen Faux */

define(FALSE,0)

/* --------------- Macros --------------- */

/* Macro de declaration des compteurs et des drapeaux de
   synchronisation, a inserer dans la partie declarative de chaque
   cellule du reseau */

define(DeclareCompteurs,
  /* Compteur de vecteur (d'apprentissage ou de test) */
  O CV;

  /* Compteur d'etape (cycle MCV) */
  O CE;

  /* Compteur de cycle general (apprentissage+test) */
  O CG;

  /* Drapeau d'apprentissage des exemples */
  O FAE;

  /* Drapeau de test des exemples */
  O FTE;

  /* Drapeau de test des inconnus */
  O FTI;

  /* Drapeau de gestion de l'ensemble */
  O FTG;

  /* Compteur des cycles sans tests */
  O CST
)

/* Macro d'initialisation des compteurs et des drapeaux de
   synchronisation, a inserer dans toutes les regles initially des
   cellules du reseau (cas special: l'afficheur, il faut mettre son
   compteur d'etape a -1) */

define(InitCompteurs,
      /* On prend le premier vecteur (numero zero) */
      CV ! 0;
```

```
      /* premiere etape de l'apprentissage */
      CE ! 0;

      /* Premier cycle general (apprentissage + test) */
      CG ! 0;

      /* Drapeau d'apprentissage des exemples */
      FAE ! TRUE;

      /* Drapeau de test des exemples */
      FTE ! FALSE;

      /* Drapeau de test des inconnus */
      FTI ! FALSE;

      /* Drapeau de Gestion de l'ensemble */
      FTG ! FALSE;

      /* Compteurs des cycles sans tests */
      CST ! 0
)


/* Macro permettant l'affectation "simultanee" d'une variable
   intermediaire et d'un canal de sortie avec la meme valeur.
   ATTENTION: la variable intermediaire doit etre declaree et avoir le
   meme nom que celui du canal de sortie precede de i (en minuscule,
   pour variable Intermediaire) */

define(affecte, i$1 ! $2; $1 ! i$1)


/* Fonction de seuil du reseau, ici c'est la sigmoide, qui donne un
   resultat entre 0 et 1 */

define(seuil, 1/(1+exp (-$1)))



/* Fonction renvoyant une valeur aleatoire (en supposant ici que le
   parametre a pour valeur a) entre -a et +a (avec une precision d'une
   seule decimale) */

define(aleatoire, ((rand (1+2*$1*10)-$1*10)/10) )


/* Macro de declaration des canaux d'entree pour la cellule appelante.
   Declare des tableaux pour la reception des resultats des reseaux
   (Parametre: Nombre de reseaux maxi) */

define(DeclareReseaux,
  /* Tables des resultats du (des) reseau(x) construit(s) */
  ? NNCycle[NBRESEAUX];
```

```
  ? NNErreurEx[NBRESEAUX];
  ? NNReconnuEx[NBRESEAUX];
  ? NNErreurInc[NBRESEAUX];
  ? NNReconnuInc[NBRESEAUX];
  ? NNFini[NBRESEAUX];
  /* Pointeur du GnlReader */
  @ GnlR;
  /* Drapeaux de lecture des resultats */
  O NNVu[NBRESEAUX];
  /* Compteur de boucle au nom bizarre */
  O NNijkz;
  /* Compteur du dernier cycle ou le reseau a produit un resultat */
  O NNCC[NBRESEAUX]
)

/* Macro de passage de l'adresse de la cellule courante */
define(Self,as_value(self))

/* Macro d'affichage des resultats intermediaires du travail d'un */
/* reseau (en cours d'apprentissage, avant le dernier cycle) :    */
/* AfficheRes(reseau)                                             */
define(AfficheRes,
      writes ("Reseau "); writev ($1);
      writes ("\tCycle "); writev (NNCycle[$1]); writes ("\n");
      writes (" Erreur exemples:\t"); writev (NNErreurEx[$1]);
      writes (" Reconnaissance:\t"); writev (NNReconnuEx[$1]);
      writes ("%\n Erreurs inconnus\t"); writev (NNErreurInc[$1]);
      writes (" Reconnaissance:\t"); writev (NNReconnuInc[$1]);
      writes ("%\n");
      if NNFini[$1]=TRUE then NNVu[$1]!TRUE;
      NNCC[$1] ! NNCycle[$1]
)


/* Macro de test de l'arrivee des resultats intermediaires */
define(Teste, (NNCycle[$1]=NNCC[$1]+NBRAPIDES+1 and NNVu[$1]=FALSE))

/* Macro de test de la fin du calcul d'un reseau */
define(Fin, (NNFini[$1]=TRUE and NNVu[$1]=FALSE))

/* Macro de fourniture d'un vecteur a un reseau. argument: le numero
   du reseau dans la cellule GnlReader */

define(FournitVecteur,
 if (Numero[$1]<NBVECTEURS or Numero[$1]>=0) and Existe[$1]=TRUE then {
   for ijk ! 0 ; to ijk >= TailleEntree by ijk ! ijk + 1 ; do
     Entree[$1][ijk] ! Entr[Numero[$1]][ijk];

   for ijk ! 0 ; to ijk >= TailleSortie by ijk ! ijk + 1 ; do
     Desire[$1][ijk] ! Desir[Numero[$1]][ijk];

   Dernier[$1] ! Numero[$1];
 };
```

```
)


/* Macro qui donne l'adresse scalaire du pointeur de GnlR */

define(GR,as_value(GnlR))


/* Macro qui initialise les drapeaux Vu dans la cellule appelante */

define(InitVus, for NNijkz!0; to NNijkz>=NBRESEAUX by NNijkz!NNijkz+1; do {
       NNVu[NNijkz]!FALSE; NNCC[NNijkz]!0; })


/*
   =======================================================
   =============== Cellule NeuralNet
   ======================================================= */
actor NeuralNet (NbCouches, TailleEntree,
                 TailleCachee1, TailleCachee2, TailleCachee3, TailleSortie,
                 NbCycles, NbEx, NbTests,
                 Seuil, Etha, Borne, Numero, Appelante,GnlReader) {

/* La synchronisation est decalee d'avec les autres cellules du
   cellules du reseau (NeuralNet est en avance d'une etape), NeuralNet
   peut ainsi communiquer au GnlReader la requete du nouveau vecteur a
   temps. */


    /* --------------- Pointeurs de cellule --------------- */

    /* Pointeur de l'afficheur */
    @ MyPrinter;

    /* Pointeurs des neurones de la premiere couche cachee */
    @ HN1[MAXITAB];

    /* Pointeurs des neurones des autres couches cachees */
    @ HN[2][MAXITAB];

    /* Pointeurs des neurones de la couche de sortie */
    @ ON[MAXITAB];

    /* Pointeur du lecteur */
    @ MyReader;

    /* --------------- Canaux de sortie --------------- */

    /* Numero du vecteur a demander au GnlReader */
    ! Vecteur;

    /* Drapeau d'existence du reseau (a transmettre au GnlReader) */
    ! Existe;
```

```
  /* --------------- Variables --------------- */
  /* Taille des couches */
  O Taille[5];

  /* Compteurs de boucle */
  O Neurone, i, Couche;

  /* Declaration des compteurs et des drapeaux de synchronisation */
  DeclareCompteurs;

rules
  /* ***** Initialisation et acouchements des cellules du reseau */
  initially ==> {

    if (NbEx+NbTests>NBVECTEURS) then {
      writes("Reseau no");
      writev(Numero);
      writes (": ERREUR dans les parametres\n");
      writes ("Trop de d'exemples et de tests pour le nombre de
    vecteurs!\n");
      writes ("Les resultats fournis seront errones!\n");
    };

    /* Recuperation des tailles des couches --------------- */
    Taille[0] ! TailleEntree;
    Taille[1] ! TailleCachee1;
    Taille[2] ! TailleCachee2;
    Taille[3] ! TailleCachee3;
    Taille[NbCouches-1] ! TailleSortie;

    /* Transformation de la valeur transmise comme adresse du
       GnlReader en un pointeur de cellule */
    define(MyReader,as_actor(GnlReader))

    /* Naissance de la couche cachee numero 1 --------------- */
    for Neurone ! 0; to Neurone >= Taille[1] by Neurone!Neurone+1; do
      HN1[Neurone] born Hidden1 (NbCouches, TailleEntree, Taille[2],
    NbCycles,
                             NbEx, NbTests, Etha, Borne);

    /* Naissance des autres couches cachees --------------- */
    for Couche ! 2; to Couche >= NbCouches-1 by Couche ! Couche + 1; do
      for Neurone ! 0; to Neurone >= Taille[Couche] by Neurone ! Neurone + 1;
      do
        HN[Couche-2][Neurone] born Hidden (NbCouches, Taille[Couche-1],
                                    Taille[Couche+1], Couche,
                                    NbCycles, NbEx, NbTests,
                                    Etha, Borne);


    /* Naissance de la couche de sortie --------------- */
    for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
```

```
  ON[Neurone] born Output (NbCouches, Taille[NbCouches-2], NbCycles,
                           NbEx, NbTests, Etha, Borne);

/* Naissance de l'afficheur --------------- */
MyPrinter born MyPrinter (NbCouches, TailleEntree, TailleCachee1,
                          TailleCachee2, TailleCachee3, TailleSortie,
                          NbCycles, NbEx, NbTests, Seuil, Etha, Borne);

/* ---------- Connexions ---------- */

/* Connexions MyPrinter-->Appelante */
connect NumCycle in MyPrinter oftype MyPrinter
  to NNCycle[Numero] in as_actor(Appelante) oftype main;

connect ErreurEx in MyPrinter oftype MyPrinter
  to NNErreurEx[Numero] in as_actor(Appelante) oftype main;

connect ReconnuEx in MyPrinter oftype MyPrinter
  to NNReconnuEx[Numero] in as_actor(Appelante) oftype main;

connect ErreurInc in MyPrinter oftype MyPrinter
  to NNErreurInc[Numero] in as_actor(Appelante) oftype main;

connect ReconnuInc in MyPrinter oftype MyPrinter
  to NNReconnuInc[Numero] in as_actor(Appelante) oftype main;

connect Fini in MyPrinter oftype MyPrinter
  to NNFini[Numero] in as_actor(Appelante) oftype main;

/* Connexions GnlReader-->Hidden */

for Neurone ! 0; to Neurone >= Taille[1] by Neurone!Neurone+1; do
  for i ! 0; to i >= TailleEntree by i ! i + 1; do
  connect Entree[Numero][i] in MyReader oftype GnlReader
    to Entree[i] in HN1[Neurone] oftype Hidden1;

/* Connexions GnlReader-->Output */

for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
  connect Desire[Numero][Neurone] in MyReader oftype GnlReader
    to Desir in ON[Neurone] oftype Output;

if NbCouches=3 then {
  /* Connexions Hidden1-->Output */
  for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
    for i ! 0 ; to i >= Taille[1] by i ! i + 1; do
      connect Sortie in HN1[i] oftype Hidden1
        to Entree[i] in ON[Neurone] oftype Output;

  /* Connexions Output-->Hidden1 */
  for Neurone ! 0; to Neurone >= Taille[1] by Neurone!Neurone+1; do
    for i ! 0; to i >= TailleSortie by i ! i + 1; do {
      connect Delta in ON[i] oftype Output
```

```
        to Delt[i] in HN1[Neurone] oftype Hidden1;

      connect Poids[Neurone] in ON[i] oftype Output
        to P[i] in HN1[Neurone] oftype Hidden1;
    };
};

if NbCouches=4 then {
  /* Connexions Hidden1-->Hidden */
  for Neurone ! 0; to Neurone >= Taille[2] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[1] by i ! i + 1; do
      connect Sortie in HN1[i] oftype Hidden1
        to Entree[i] in HN[0][Neurone] oftype Hidden;

  /* Connexions Hidden-->Hidden1 */
  for Neurone ! 0; to Neurone >= Taille[1] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[2] by i ! i + 1; do{
      connect Delta in HN[0][i] oftype Hidden
        to Delt[i] in HN1[Neurone] oftype Hidden1;

      connect Poids[Neurone] in HN[0][i] oftype Hidden
        to P[i] in HN1[Neurone] oftype Hidden1;
    };

  /* Connexions Hidden-->Output */
  for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
    for i ! 0 ; to i >= Taille[2] by i ! i + 1; do
      connect Sortie in HN[0][i] oftype Hidden
        to Entree[i] in ON[Neurone] oftype Output;

  /* Connexions Output-->Hidden */
  for Neurone ! 0; to Neurone >= Taille[2] by Neurone!Neurone+1; do
    for i ! 0; to i >= TailleSortie by i ! i + 1; do {
      connect Delta in ON[i] oftype Output
        to Delt[i] in HN[0][Neurone] oftype Hidden;

      connect Poids[Neurone] in ON[i] oftype Output
        to P[i] in HN[0][Neurone] oftype Hidden;
    };
};

if NbCouches=5 then {
  /* Connexions Hidden1-->Hidden2*/
  for Neurone ! 0; to Neurone >= Taille[2] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[1] by i ! i + 1; do
      connect Sortie in HN1[i] oftype Hidden1
        to Entree[i] in HN[0][Neurone] oftype Hidden ;

  /* Connexions Hidden2-->Hidden3 */
  for Neurone ! 0; to Neurone >= Taille[3] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[2] by i ! i + 1; do
      connect Sortie in HN[0][i] oftype Hidden
        to Entree[i] in HN[1][Neurone] oftype Hidden ;
```

```
  /* Connexions Hidden3-->Hidden2 */
  for Neurone ! 0; to Neurone >= Taille[2] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[3] by i ! i + 1; do{
      connect Delta in HN[1][i] oftype Hidden
        to Delt[i] in HN[0][Neurone] oftype Hidden;

      connect Poids[Neurone] in HN[1][i] oftype Hidden
        to P[i] in HN[0][Neurone] oftype Hidden;
    };

  /* Connexions Hidden2-->Hidden1 */
  for Neurone ! 0; to Neurone >= Taille[1] by Neurone!Neurone+1; do
    for i ! 0; to i >= Taille[2] by i ! i + 1; do{
      connect Delta in HN[0][i] oftype Hidden
        to Delt[i] in HN1[Neurone] oftype Hidden1;

      connect Poids[Neurone] in HN[0][i] oftype Hidden
        to P[i] in HN1[Neurone] oftype Hidden1;
    };

  /* Connexions Hidden3-->Output */
  for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
    for i ! 0 ; to i >= Taille[3] by i ! i + 1; do
      connect Sortie in HN[1][i] oftype Hidden
        to Entree[i] in ON[Neurone] oftype Output;

  /* Connexions Output-->Hidden3 */
  for Neurone ! 0; to Neurone >= Taille[3] by Neurone!Neurone+1; do
    for i ! 0; to i >= TailleSortie by i ! i + 1; do {
      connect Delta in ON[i] oftype Output
        to Delt[i] in HN[1][Neurone] oftype Hidden;

      connect Poids[Neurone] in ON[i] oftype Output
        to P[i] in HN[1][Neurone] oftype Hidden;
    };
};

/* Connexions Output-->MyPrinter */
for Neurone ! 0; to Neurone >= TailleSortie by Neurone!Neurone+1; do
  connect Erreur in ON[Neurone] oftype Output
    to Erreur[Neurone] in MyPrinter oftype MyPrinter;

/* Connexions NeuralNet --> GnlReader */
connect Vecteur in self oftype NeuralNet
  to Numero[Numero] in MyReader oftype GnlReader;

connect Existe in self oftype NeuralNet
  to Existe[Numero] in MyReader oftype GnlReader;

/* Initialisation des compteurs et des drapeaux de
   synchronisation */
InitCompteurs;
```

```
    /* On signale au reader que le reseau existe */
    Existe ! TRUE;

    Vecteur ! 0;

  }; /* ***** */

  /* ***** Phase d'apprentissage des exemples */
  FAE = TRUE ==> {

    /* Passage a l'etape suivante */
    CE ! CE + 1;

    /* Passage au vecteur suivant */
    if CE = 2*NbCouches-1 then {
      CE ! 0;
      CV ! CV + 1;
      Vecteur ! CV;
    };

    /* Passage a la phase de test des exemples */
    if CV = NbEx and CST=NBRAPIDES then {
      CV ! 0;
      CE ! 0;
      FAE ! FALSE;
      FTE ! TRUE;
      Vecteur ! CV;
    };
    else
      /* Passage a la phase de gestion des cycles generaux */
      if CV = NbEx then {
        CV ! 0;
        CE ! 0;
        FAE ! FALSE;
        FTG ! TRUE;
      };

  }; /* ***** */

  /* Phase de test des exemples */
  FTE = TRUE ==> {

    /* Passage a l'etape suivante */
    CE ! CE + 1;

    /* Passage au vecteur suivant */
    if CE = NbCouches then {
      CE ! 0;
      CV ! CV + 1;
      Vecteur ! CV;
    };
```

```
  /* Passage a la phase de test des inconnus */
  if CV = NbEx then {
    CE ! 0;
    FTE ! FALSE;
    FTI ! TRUE;
    Vecteur ! CV;
  };

}; /* ***** */


/* Phase de test des inconnus */
FTI = TRUE ==> {

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
    Vecteur ! CV;
  };

  /* Passage a la phase de gestion des cycles generaux */
  if CV = NbEx+NbTests then {
    CV ! 0;
    CE ! 0;
    FTI ! FALSE;
    FTG ! TRUE;
    Vecteur ! CV;
  };

}; /* ***** */

/* ***** Phase de gestion des cycles generaux */
FTG = TRUE ==> {

  /* Passage au cycle suivant */
  CG ! CG + 1;
  CST ! CST + 1;
  if CST = NBRAPIDES+1 then CST ! 0;

  if CG < NbCycles then {
    FAE ! TRUE;
    FTG ! FALSE;
  };
  else {
    self dead;
  };

}; /* ***** */
```

```
}; /* ============================== */


/* ======================================================
   =============== Cellule GnlReader
   ====================================================== */
actor GnlReader (TailleEntree, TailleSortie) {

  /* --------------- Canaux de sortie --------------- */

  /* Table du vecteur d'entree */
  ! Entree[NBRESEAUX][MAXITAB];

  /* Table du vecteur desire */
  ! Desire[NBRESEAUX][MAXITAB];

  /* --------------- Canaux d'entree --------------- */

  /* table du Numero du vecteur demande par les reseaux */
  ? Numero[NBRESEAUX];

  /* Drapeau d'existence des reseaux possibles */
  ? Existe[NBRESEAUX];

  /* --------------- Variables --------------- */

  /* Table des vecteurs d'entree */
  0 Entr[NBVECTEURS][MAXITAB];

  /* Table des vecteurs desires */
  0 Desir[NBVECTEURS][MAXITAB];

  /* Compteurs de boucle */
  0 Num, i, ijk, Chiffre;

  /* Derniers numeros demandes par les reseaux */
  0 Dernier[NBRESEAUX];

  rules
    /* ***** Lecture des donnees et generation des vecteurs desires */
    initially ==>{

      /* Lecture des vecteurs d'entree
         Le programme est classiquement lance par: net <data.in
         car il recoit ses entrees par le port d'entree standard */
      for Num ! 0; to Num >= NBVECTEURS by Num ! Num + 1; do
        for i ! 0; to i >= TailleEntree by i ! i + 1; do
          readv (Entr[Num][i]);

      /* Generation des vecteurs desires */
      Chiffre ! 0;
      for Num ! 0; to Num >= NBVECTEURS by Num ! Num + 1; do {
        for i ! 0; to i >= TailleSortie by i ! i + 1; do
```

```
          if Chiffre = i then Desir[Num][i] ! 1; else Desir[Num][i] ! 0;

        if Chiffre = 9 then Chiffre ! 0; else Chiffre ! Chiffre + 1;
      };

      /* Initialisation des demandes des reseaux a rien */
      for i ! 0 ; to i >= NBRESEAUX by i ! i + 1 ; do
        Dernier[i] ! -1;

    }; /* ***** */

    /* ***** fourniture d'un vecteur a un reseau */
    (Numero[0]<>Dernier[0] or Numero[1]<>Dernier[1]) ==> {

      for i ! 0; to i >= NBRESEAUX by i ! i + 1; do
        if Numero[i]<>Dernier[i] then
          FournitVecteur(i);

    }; /* ***** */

}; /* ============================== */



/* ==========================================================
   =============== Cellule MyPrinter
   ========================================================== */
actor MyPrinter (NbCouches,
                 TailleEntree, TailleCachee1, TailleCachee2, TailleCachee3,
                 TailleSortie, NbCycles, NbEx, NbTests, Seuil, Etha, Borne) {

  /* --------------- Canaux de sortie --------------- */

  /* Numero du cycle courant (pour les derniers resultats uniquement) */
  ! NumCycle;

  /* Erreur des exemples pour le dernier cycle */
  ! ErreurEx;

  /* Taux de reconnaissance des exemples pour le dernier cycle */
  ! ReconnuEx;

  /* Erreur des inconnus pour le dernier cycle */
  ! ErreurInc;

  /* Taux de reconnaissance des inconnus pour le dernier cycle */
  ! ReconnuInc;

  /* Drapeau de fin des operations pour le reseau */
  ! Fini;

  /* --------------- Canaux d'entree --------------- */
```

```
/* Erreurs quadratiques des neurones de la couche de sortie */
? Erreur[MAXITAB];

/* -------------- Variables -------------- */

/* Declaration des compteurs et des drapeaux de synchronisation */
DeclareCompteurs;

/* Erreur locale au vecteur courant */
O ErreurVecteur;

/* Erreur moyenne d'une phase */
O ErreurMoyenne;

/* Pourcentage de reconnaissance des vecteurs d'une phase */
O Pourcentage;

/* Taille des couches */
O Taille[5];

/* Compteur de boucle */
O Neurone, Couche;

/* Exemple: Erreur Moyenne (variable intermediaire) */
O EEM;

/* Exemple: Pourcentage (variable intermediaire) */
O EP;

rules
  /* ***** Initialisation */
  initially ==> {

    /* Recuperation des tailles des couches -------------- */
    Taille[0] ! TailleEntree;
    Taille[1] ! TailleCachee1;
    Taille[2] ! TailleCachee2;
    Taille[3] ! TailleCachee3;
    Taille[NbCouches-1] ! TailleSortie;

    writes("'#' Reseau de neurones a retropropagation du gradient\n");
    writes ("'#' "); writev (NbCouches); writes(" couches:\n");
    writes ("'#' Couche d'entree:\t\t"); writev(TailleEntree);
    writes (" neurones\n");
    for Couche ! 1; to Couche >= NbCouches-1 by Couche ! Couche + 1; do {
      writes ("'#' Couche cachee ");writev (Couche);
      writes (":\t"); writev(Taille[Couche]); writes(" neurones\n");
    };
    writes ("'#' Couche de sortie:\t\t"); writev(TailleSortie);
    writes ("  neurones\n");
    writes ("'#' "); writev(NbCycles); writes(" cycles "); writev(NbEx);
    writes(" exemples "); writev(NbTests); writes(" tests\n");
    writes ("'#' coef "); writev(Etha); writes(", Seuil "); writev(Seuil);
```

```
  writes(", borne "); writev(Borne); writes("\n\n");

  /* Initialisation des compteurs et des drapeaux de
     synchronisation */
  InitCompteurs;

  /* Comme nous sommes dans le Printer, nous allons decaler le
     compteur d'etape, ainsi le Printer pourra afficher a son
     etape N les sorties des autres cellules a l'etape N
     egalement. */
  CE ! -1;

  /* Initialisation des variables statistiques */
  ErreurMoyenne ! 0;
  Pourcentage ! 0;

  /* Initialisation des resultats de sortie */
  NumCycle   ! -1;
  ErreurEx   ! -1;
  ReconnuEx  ! -1;
  ErreurInc  ! -1;
  ReconnuInc ! -1;
  Fini       ! FALSE;

}; /* ***** */

/* ***** Phase d'apprentissage des exemples */
FAE = TRUE ==> {

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = 2*NbCouches-1 then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des exemples */
  if CV = NbEx and CST=NBRAPIDES then {
    CV ! 0;
    CE ! 0;
    FAE ! FALSE;
    FTE ! TRUE;

    /* Initialisation pour la phase suivante */
    ErreurMoyenne ! 0;
    Pourcentage ! 0;
  };
  else
    /* Passage a la phase de gestion des cycles generaux */
    if CV = NbEx then {
      CV ! 0;
```

```
      CE ! 0;
      FAE ! FALSE;
      FTG ! TRUE;
    };

};  /* ***** */

/* Phase de test des exemples */
FTE = TRUE ==> {


  /* Propagation Output --> Printer */
  if CE = NbCouches-1 then {

    ErreurVecteur ! 0;

    for Neurone ! 0; to Neurone >= TailleSortie by Neurone ! Neurone+1; do
      ErreurVecteur ! ErreurVecteur + Erreur[Neurone];

    ErreurMoyenne ! ErreurMoyenne + ErreurVecteur;

    if ErreurVecteur < Seuil then
      Pourcentage ! Pourcentage + 1;
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des inconnus */
  if CV = NbEx then {
    CE ! 0;
    FTE ! FALSE;
    FTI ! TRUE;

    /* Affichage de l'erreur moyenne des tests des exemples */
    ErreurMoyenne ! ErreurMoyenne/NbEx;
    Pourcentage ! 100 * Pourcentage / NbEx;

    EEM ! ErreurMoyenne;
    EP  ! Pourcentage;

    /* Initialisation pour la phase suivante */
    ErreurMoyenne ! 0;
    Pourcentage ! 0;
  };

};  /* ***** */
```

```
/* Phase de test des inconnus */
FTI = TRUE ==> {


  /* Propagation Output --> Printer */
  if CE = NbCouches-1 then {

    ErreurVecteur ! 0;

    for Neurone ! 0; to Neurone >= TailleSortie by Neurone ! Neurone+1; do
      ErreurVecteur ! ErreurVecteur + Erreur[Neurone];

    ErreurMoyenne ! ErreurMoyenne + ErreurVecteur;

    if ErreurVecteur < Seuil then
      Pourcentage ! Pourcentage + 1;
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de gestion des cycles generaux */
  if CV = NbEx+NbTests then {
    CV ! 0;
    CE ! 0;
    FTI ! FALSE;
    FTG ! TRUE;

    /* Affichage de l'erreur moyenne des tests des exemples */
    ErreurMoyenne ! ErreurMoyenne/NbTests;
    Pourcentage ! 100 * Pourcentage / NbTests;

    /* Passage des resultats a l'Appelante */
    NumCycle ! CG;
    ErreurEx ! EEM;
    ReconnuEx ! EP;
    ErreurInc ! ErreurMoyenne;
    ReconnuInc! Pourcentage;

    /* Initialisation pour la phase suivante */
    ErreurMoyenne ! 0;
    Pourcentage ! 0;
  };

}; /* ***** */
```

```
    /* ***** Phase de gestion des cycles generaux */
    FTG = TRUE ==> {

      /* Information a l'Appelante : numero du cycle passe */
      NumCycle ! CG ;
      /* Passage au cycle suivant */
      CG ! CG + 1;
      if CST = NBRAPIDES then CST ! 0; else CST ! CST + 1;

      if CG < NbCycles then {
        FAE ! TRUE;
        FTG ! FALSE;
      };
      else {
        Fini ! TRUE;
        self dead;
      };

    }; /* ***** */

}; /* ============================== */



/* =======================================================
   =============== Cellule Hidden1
   ======================================================= */
actor Hidden1(NbCouches, TailleEntree, TailleSuiv,
              NbCycles, NbEx, NbTests, Etha, Borne) {

  /* --------------- Canaux d'entree --------------- */

  /* Valeurs des entrees du neurone */
  ? Entree[MAXITAB];

  /* Signaux d'erreur de la couche de sortie */
  ? Delt[MAXITAB];

  /* Poids des connexions avec la couche de sortie */
  ? P[MAXITAB];

  /* --------------- Canaux de sortie --------------- */

  /* Valeur de sortie du neurone */
  ! Sortie;

  /* --------------- Variables --------------- */

  /* Declaration des compteurs et des drapeaux de synchronisation */
  DeclareCompteurs;

  /* Poids des connexions */
```

```
O W[MAXITAB];

/* Valeur du signal d'erreur du neurone */
O Delta;

O Total, i, Somme;

/* Variables intermediaires */
O iSortie;

rules
  /* ***** Initialisation des poids des connexions */
  initially ==> {

    /* Initialisation des compteurs de synchronisation */
    InitCompteurs;

    /* Initialisation des poids des connexions du neurone */
    for i ! 0; to i >= TailleEntree by i ! i + 1; do
      W[i] ! aleatoire(Borne);

  }; /* ***** */

  /* ***** Phase d'apprentissage des exemples */
  FAE = TRUE ==> {

    /* Propagation Reader-->Hidden */
    if CE = 1 then {

      Total ! 0;

      for i ! 0; to i >= TailleEntree by i ! i + 1; do
        Total ! Total + Entree[i] * W[i];

      /* Seuillage */
      affecte(Sortie,seuil(Total));
    };

    /* Retropropagation Hidden1 -->Reader */
    if CE = 2*NbCouches-2 then {

      Somme ! 0;

      for i ! 0; to i >= TailleSuiv by i ! i + 1; do
        Somme ! Somme + Delt[i] * P[i];

      Delta ! iSortie * (1-iSortie) * Somme;

      /* Modifications des poids des connexions avec la couche
         precedente */
      for i ! 0; to i >= TailleEntree by i ! i + 1; do
        W[i] ! W[i] + Etha * Delta * Entree[i];
```

```
  };


  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = 2*NbCouches-1 then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des exemples */
  if CV = NbEx and CST=NBRAPIDES then {
    CV ! 0;
    CE ! 0;
    FAE ! FALSE;
    FTE ! TRUE;
  };
  else
    /* Passage a la phase de gestion des cycles generaux */
    if CV = NbEx then {
      CV ! 0;
      CE ! 0;
      FAE ! FALSE;
      FTG ! TRUE;
    };

}; /* ***** */

/* ***** Phase de test des exemples */
FTE = TRUE ==>{

  /* Propagation Reader-->Hidden */
  if CE = 1 then {

    Total ! 0;

    for i ! 0; to i >= TailleEntree by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    affecte(Sortie,seuil(Total));
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };
```

```
  /* Passage a la phase de test des inconnus */
  if CV = NbEx then {
    CE ! 0;
    FTE ! FALSE;
    FTI ! TRUE;
  };

}; /* ***** */


/* ***** Phase de test des inconnus */
FTI = TRUE ==>{

  /* Propagation Reader-->Hidden */
  if CE = 1 then {

    Total ! 0;

    for i ! 0; to i >= TailleEntree by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    affecte(Sortie,seuil(Total));
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de gestion des cycles generaux */
  if CV = NbEx+NbTests then {
    CV ! 0;
    CE ! 0;
    FTI ! FALSE;
    FTG ! TRUE;
  };

}; /* ***** */

/* ***** Phase de gestion des cycles generaux */
FTG = TRUE ==> {

  /* Passage au cycle suivant */
  CG ! CG + 1;
  if CST=NBRAPIDES then CST ! 0; else CST ! CST + 1;

  /* S'il reste des cycles a faire */
```

```
      if CG < NbCycles then {
        FAE ! TRUE;
        FTG ! FALSE;
      };
      else
        self dead;

    }; /* ***** */

}; /* ============================== */



/* =======================================================
   =============== Cellule Hidden
   ======================================================= */
actor Hidden (NbCouches, TaillePrec, TailleSuiv, Couche,
              NbCycles, NbEx, NbTests, Etha, Borne) {

  /* --------------- Canaux d'entree --------------- */

  /* Valeurs des entrees du neurone */
  ? Entree[MAXITAB];

  /* Signaux d'erreur de la couche suivante */
  ? Delt[MAXITAB];

  /* Poids des connexions avec la couche suivante */
  ? P[MAXITAB];

  /* --------------- Canaux de sortie --------------- */

  /* Valeur de sortie du neurone */
  ! Sortie;

  /* Signal d'erreur du neurone, transmise aux neurones de la couche
     cachee precedente */
  ! Delta;

  /* Poids des connexions avec la couche precedente, qui en a besoin
     lors de la retropropagation */
  ! Poids[MAXITAB];

  /* --------------- Variables --------------- */

  /* Declaration des compteurs et des drapeaux de synchronisation */
  DeclareCompteurs;

  /* Poids des connexions */
  O W[MAXITAB];

  /* Valeur du signal d'erreur du neurone */
  O iDelta;
```

```
O Total, i, Somme;

/* Variables intermediaires */
O iSortie;

rules
  /* ***** Initialisation des poids des connexions */
  initially ==> {

    /* Initialisation des compteurs de synchronisation */
    InitCompteurs;

    /* Initialisation des poids des connexions du neurone */
    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      W[i] ! aleatoire(Borne);

  }; /* ***** */

  /* ***** Phase d'apprentissage des exemples */
  FAE = TRUE ==> {

    /* Propagation Hidden-->Hidden2 */
    if CE = Couche then {

      Total ! 0;

      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        Total ! Total + Entree[i] * W[i];

      /* Seuillage */
      affecte(Sortie,seuil(Total));
    };

    /* Retropropagation Hidden --> Couche precedente */
    if CE = NbCouches+3-Couche then {

      Somme ! 0;

      for i ! 0; to i >= TailleSuiv by i ! i + 1; do
        Somme ! Somme + Delt[i] * P[i];

      affecte(Delta,iSortie * (1-iSortie) * Somme);

      /* Transmission des poids des connexions a la couche
         precedente */
      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        Poids[i] ! W[i];

      /* Modifications des poids des connexions avec la couche
         precedente */
      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        W[i] ! W[i] + Etha * iDelta * Entree[i];
```

```
  };


  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = 2*NbCouches-1 then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des exemples */
  if CV = NbEx and CST=NBRAPIDES then {
    CV ! 0;
    CE ! 0;
    FAE ! FALSE;
    FTE ! TRUE;
  };
  else
    /* Passage a la phase de gestion des cycles generaux */
    if CV = NbEx then {
      CV ! 0;
      CE ! 0;
      FAE ! FALSE;
      FTG ! TRUE;
    };

}; /* ***** */

/* ***** Phase de test des exemples */
FTE = TRUE ==>{

  /* Propagation Hidden --> couche suivante */
  if CE = Couche then {

    Total ! 0;

    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    affecte(Sortie,seuil(Total));
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
```

```
  };

  /* Passage a la phase de test des inconnus */
  if CV = NbEx then {
    CE ! 0;
    FTE ! FALSE;
    FTI ! TRUE;
  };

}; /* ***** */


/* ***** Phase de test des inconnus */
FTI = TRUE ==>{

  /* Propagation couche precedente --> Hidden */
  if CE = Couche then {

    Total ! 0;

    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    affecte(Sortie,seuil(Total));
  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de gestion des cycles generaux */
  if CV = NbEx+NbTests then {
    CV ! 0;
    CE ! 0;
    FTI ! FALSE;
    FTG ! TRUE;
  };

}; /* ***** */

/* ***** Phase de gestion des cycles generaux */
FTG = TRUE ==> {

  /* Passage au cycle suivant */
  CG ! CG + 1;
  if CST=NBRAPIDES then CST ! 0; else CST ! CST + 1;
```

```
      /* S'il reste des cycles a faire */
      if CG < NbCycles then {
        FAE ! TRUE;
        FTG ! FALSE;
      };
      else
        self dead;

    }; /* ***** */

}; /* ============================== */



/* =======================================================
   =============== Cellule Output
   ======================================================= */
actor Output (NbCouches, TaillePrec, NbCycles, NbEx, NbTests, Etha, Borne) {

  /* --------------- Canaux d'entree --------------- */

  /* Valeurs des entrees du neurone */
  ? Entree[MAXITAB];

  /* Valeur desiree du vecteur en sortie */
  ? Desir;

  /* --------------- Canaux de sortie --------------- */

  /* Erreur Quadratique du neurone */
  ! Erreur;

  /* Valeur du signal d'erreur du neurone transmise aux neurones de la
     couche cachee */
  ! Delta;

  /* Poids des connexions de la couche avec la couche cachee, que l'on
     transmet a cette couche cachee pour la retropropagation */
  ! Poids[MAXITAB];

  /* --------------- Variables --------------- */

  /* Declaration des compteurs et des drapeaux de synchronisation */
  DeclareCompteurs;

  /* Poids des connexions */
  O W[MAXITAB];

  /* Valeur de sortie du neurone */
  O Sortie;

  O Total, i;
```

```
/* Variables intermediaires */
O iDelta;

rules
  /* ***** Initialisation des poids des connexions */
  initially ==> {

    /* Initialisation des compteurs de synchronisation */
    InitCompteurs;

    /* Initialisation des poids des connexions du neurone */
    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      W[i] ! aleatoire(Borne);

  }; /* ***** */

  /* ***** Phase d'apprentissage des exemples */
  FAE = TRUE ==> {

    /* Propagation Couche precedente --> Output */
    if CE = NbCouches-1 then {

      Total ! 0;

      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        Total ! Total + Entree[i] * W[i];

      /* Seuillage */
      Sortie ! seuil(Total);

      /* Calcul et envoi de l'erreur */
      Erreur ! ((Sortie-Desir)*(Sortie-Desir));

    };

    /* Retropropagation Output --> couche precedente */
    if CE = NbCouches then {
      affecte(Delta, 2 * (Desir - Sortie) * Sortie * (1 - Sortie) );

      /* Transmission des poids des connexions a la couche cachee */
      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        Poids[i] ! W[i];

      /* Modification des poids des connexions avec la couche cachee */
      for i ! 0; to i >= TaillePrec by i ! i + 1; do
        W[i] ! W[i] + Etha * iDelta * Entree[i];

    };

    /* Passage a l'etape suivante */
    CE ! CE + 1;

    /* Passage au vecteur suivant */
```

```
  if CE = 2*NbCouches-1 then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des exemples */
  if CV = NbEx and CST=NBRAPIDES then {
    CV ! 0;
    CE ! 0;
    FAE ! FALSE;
    FTE ! TRUE;
  };
  else
    /* Passage a la phase de gestion des cycles generaux */
    if CV = NbEx then {
      CV ! 0;
      CE ! 0;
      FAE ! FALSE;
      FTG ! TRUE;
    };

}; /* ***** */

/* ***** Phase de test des exemples */
FTE = TRUE ==>{

  /* Propagation Hidden2-->Output */
  if CE = NbCouches-1 then {

    Total ! 0;

    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    Sortie ! seuil(Total);

    /* Calcul et envoi de l'erreur */
    Erreur ! ((Sortie-Desir)*(Sortie-Desir));

  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de test des inconnus */
  if CV = NbEx then {
```

```
    CE ! 0;
    FTE ! FALSE;
    FTI ! TRUE;
  };

}; /* ***** */


/* ***** Phase de test des inconnus */
FTI = TRUE ==>{

  /* Propagation Hidden2-->Output */
  if CE = NbCouches-1 then {

    Total ! 0;

    for i ! 0; to i >= TaillePrec by i ! i + 1; do
      Total ! Total + Entree[i] * W[i];

    /* Seuillage */
    Sortie ! seuil(Total);

    /* Calcul et envoi de l'erreur */
    Erreur ! ((Sortie-Desir)*(Sortie-Desir));

  };

  /* Passage a l'etape suivante */
  CE ! CE + 1;

  /* Passage au vecteur suivant */
  if CE = NbCouches then {
    CE ! 0;
    CV ! CV + 1;
  };

  /* Passage a la phase de gestion des cycles generaux */
  if CV = NbEx+NbTests then {
    CV ! 0;
    CE ! 0;
    FTI ! FALSE;
    FTG ! TRUE;
  };

}; /* ***** */

/* ***** Phase de gestion des cycles generaux */
FTG = TRUE ==> {

  /* Passage au cycle suivant */
  CG ! CG + 1;
  if CST=NBRAPIDES then CST ! 0; else CST ! CST + 1;
```

```
      /* S'il reste des cycles a faire */
      if CG < NbCycles then {
        FAE ! TRUE;
        FTG ! FALSE;
      };
      else
        self dead;

    }; /* ***** */

}; /* ============================== */
```

Annexe B

Exemple d'utilisation

```
/* ***************************************************************************
   util##.m : Utilisation de la bibliotheque de reseau net.m
   ***************************************************************************

   Revision  date   Commentaires
   11       29/7/93

                    Utilisation de la nouvelle bibliotheque paper.m

   ***************************************************************************/

define(NBRAPIDES,0)
include(paper.m)


/* =======================================================
   =============== Cellule Principale
   ======================================================= */
actor main{
  DeclareReseaux;

  rules
    /* ***** Lancement des reseaux */
    initially==>{
      GnlR born GnlReader(256,10);
      born NeuralNet(3,256,100,0,0,10,50,360,120,0.2,1.0,0.5,0,Self,GR);
      born NeuralNet(4,256,100,50,0,10,50,360,120,0.2,1.0,0.5,1,Self,GR);
      InitVus;
    }; /* ***** */

    /* ***** Affichage du premier reseau */
    Teste(0)==>{
      AfficheResInt(0);
    }; /* ***** */

    /* ***** Affichage du deuxieme reseau */
    Teste(1)==>{
      AfficheResInt(1);
```

```
    }; /* ***** */
};

run;
```